



Building a C-Subset Compiler for the FRISC Architecture

From Formal Languages to Executable Code

DR. KARLO KNEŽEVIĆ

Zagreb, 2026



Building a C-Subset Compiler for the FRISC Architecture

From Formal Languages to Executable Code

DR. KARLO KNEŽEVIĆ

Zagreb, 2026

*Building a C-Subset Compiler for the FRISC Architecture:
From Formal Languages to Executable Code*

Copyright © 2026 Karlo Knežević.
All rights reserved.

No part of this book may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law. For permission requests, contact the author through the companion repository listed below.

Author: Dr. Karlo Knežević.
Self-published by the author.
Zagreb, Croatia, 2026.

ISBN 978-953-47198-0-0
DOI [10.5281/zenodo.20511074](https://doi.org/10.5281/zenodo.20511074)
Published on Zenodo: <https://doi.org/10.5281/zenodo.20511074>.

The software accompanying this book lives in two open-source repositories: the FRISCcc compiler dissected throughout the text, and the separate build-along companion project of *tinyC* starter, solution, and test modules. Both are made available under an open licence. Consult the [build-guide appendix](#) for the repository locations and build instructions.

Disclaimer. This book and the accompanying software are provided on an “as is” basis, without warranty of any kind. The author has taken care to ensure the accuracy of the material; nevertheless, neither the author nor any distributor shall be liable for any loss or damage arising directly or indirectly from the use of, or reliance on, the contents of this book or its software.

Trademarks. FRISC is an educational instruction-set architecture associated with the Faculty of Electrical Engineering and Computing, University of Zagreb. Product and company names mentioned herein may be the trademarks of their respective owners and are used for identification and educational purposes only, with no claim of affiliation or endorsement.

Colophon. Typeset by the author with LuaTeX in Latin Modern (text and mathematics) and Inconsolata (code). Body set at 10.5pt on a Crown Quarto (189 × 246 mm) trim. Figures rendered with TikZ, D2, Graphviz, matplotlib, and Asymptote; syntax-highlighted listings produced with Pygments. Cover and interior design by the author.

*For my wife Katarina,
and for our daughter Mila and son Jakov.*

*In the beginning was the Word;
then came language, meaning,
and the desire to understand:*

how does a word become a world?

Foreword

A compiler is a machine that reads programs and writes programs. It is also a teaching object that hides one of the most beautiful pieces of applied mathematics inside a command-line tool. Most working programmers treat their compiler the way most drivers treat their gearbox: as an opaque box that turns one thing into another and that you replace, not repair, when it breaks. This book takes the cover off, slowly and deliberately, on a real compiler that fits in your head.

The compiler is FRISCcc: a complete implementation of a deterministic subset of C that targets the FRISC educational ISA. It is small enough to read in a long weekend and large enough to be interesting — a lexer that builds NFAs and runs them as DFAs, a canonical LR(1) parser whose table is a hundred kilobytes of pre-computed state, a semantic analyser that decorates the parse tree with types and lifetimes, an intermediate representation that decouples the front end from the target, sixteen optimisation passes that each prove a small theorem about what they preserve, and a code generator that gives every IR temporary its own stack slot and emits assembly a 1960s machine would have been comfortable executing. None of it is a toy. The whole pipeline is roughly forty thousand lines of Java; you can read the source in a week and rebuild it in three.

The reason such a small artefact is interesting is the same reason a small mechanical clock is interesting: every part is visible, every part has a job, and the whole thing works. That is rare in software. We think it is worth slowing down to look at.

This book grew out of teaching the *Prevodenje programskih jezika* (Programming Language Translation, PLT) course at the Faculty of Electrical Engineering and Computing at the University of Zagreb, where the FRISC architecture taught generations of students what a computer actually does at the instruction-set level: for over a decade the teaching machine of the *Computer Architecture 1* course, until ARM succeeded it there in 2015, and the compilation target in the compilers course ever since. The course's lecture material lives in Croatian; this book translates the engineering into English and turns the artefact into something you can build along with, chapter by chapter, in a companion repository written in a smaller language called tinyC. By the time we have walked our anchor programs through every phase — `return 5*3`; in the first chapter, iterative Fibonacci through the front end, the Sieve of Eratosthenes through the back — the box should no longer be opaque, and you should know how to open the next one yourself.

Preface

Who this book is for

We wrote this for the programmer who has used a compiler every day for years and has never quite known what happens between `gcc foo.c` and the executable that comes back. You should be comfortable reading C and Java; you do not need to have studied formal languages, parsing theory, or computer architecture. By the time `return 5*3;` becomes seven phases of explicit transformation in [Chapter 1](#), the engineering will feel familiar and the mathematics will feel earned.

The book is also for the student standing in front of a compilers course and wondering what an LR(1) parser, a typed intermediate representation, or a peephole optimiser actually look like in code. The reference compiler is real — you can read it, build it, run it, and break it on the same machine that runs the typesetter for this book. Everything we describe is observable. Nothing is hand-waved into existence.

How this book is organised

The book has fifteen chapters in five parts. Part I (chapters 1–2) establishes what we are building and gives the shape of the journey. Part II (chapters 3–6) walks the front end: the lexer that turns characters into tokens, the parser that builds a tree from them, the semantic analyser that decorates the tree with types, and the intermediate representation we lower the tree into. Part III (chapters 7–10) handles the back end: optimisation, code generation, the FRISC runtime, and the simulator that runs the result. Part IV (chapters 11–12) takes a side road and rewrites the back end twice — as a tree-walking interpreter, then as a bytecode virtual machine — so the trade-offs between compiling, interpreting, and virtual execution become a thing you can see in the same code base. Part V (chapters 13–15) brings everything together with case studies, performance measurements, and a forward look at the techniques the book did not cover.

A complete compiler involves more formality than a narrative chapter can carry without breaking pace. Where formal grammars, typing judgements, semantic-preservation theorems, ABI tables, and proofs belong, they live in the appendices: Appendix A is a note on sources, including the Croatian-to-English non-terminal glossary for the PLT-FER course material the grammar derives from; Appendix B is the FRISC ISA reference; Appendix C documents the simulator; Appendix D is the IR grammar; Appendix E is a hands-on build guide; Appendix F holds selected solutions to the chapter exercises; and [Appendix G](#) contains the

semantic-preservation proofs for every optimisation pass. Chapter prose stays narrative; the appendices carry the formality. We read neither one as decoration.

Build along: the companion repository

The book draws on two open-source repositories, and it helps to be clear from the start about which is which. The first is the *FRISCCc compiler* itself — the complete reference compiler this book dissects, open source [47] at <https://github.com/KarloKnezevic/ccompiler>. Reading the book and reading its source side by side will get you most of the way: clone it and follow along as each chapter takes its phases apart. Building a smaller compiler of your own, chapter by chapter, will get you the rest, and that is what the second repository is for. The *companion* repository — the build-along project [46] at <https://github.com/KarloKnezevic/frisccc-companion> — contains a Maven project for a smaller language called *tinyC* — an integer-only C subset, no floats, no arrays, no pointers, no globals — and one module per chapter from Chapter 3 onward. Every module ships with a `starter/` directory of stubs marked `// TODO: implement`, a `solution/` directory you can peek at when stuck, and a `tests/` directory of JUnit cases that verify behaviour against the reference. We refer to it as the *Your turn* pointer at the end of each chapter that has a corresponding module. Use it however you like: build *tinyC* end-to-end as you read; treat it as an exercise sandbox; or skip the companion entirely and just read. The book is self-contained either way.

Conventions used in this book

The voice is first-person plural. We — author and reader together — are inside the same machine, watching the same artefacts move from one phase to the next. We avoid the documentary register and the distancing “one might note”. When we know we are guessing, we say so; when we know we are right, we say so for the same reason.

The body of each chapter stays narrative. Sidebars, marked with the *Aside* title, carry historical context, formal definitions, and extended remarks that would derail the running argument. Boxes labelled *key insight* are deliberate stopping points; if a chapter has one, it is the thing we want you to remember after the rest has faded. Boxes labelled *Warning* flag a subtle pitfall; *Engineering tip* points at a real-world refinement we use in the *FRISCCc* code.

Listings have line numbers and a coloured rule on the left. Inline code is rendered as `code{like this}`. Cross-references use `autoref`, so any reference to a chapter, section, figure, theorem, or listing comes with its kind already prefixed. Citations are numeric in square brackets and resolve to the bibliography. Algorithm boxes are numbered per chapter; the list of algorithms is at the front of the book alongside the list of figures and tables.

Croatian appears in exactly one place: the non-terminal names in the grammar file (`prijevodna_jedinica`, `aditivni_izraz`, and so on). The course material the grammar derives from is in Croatian, and renaming the file would silently break every reference to it. We translate every non-terminal name in the running prose; the complete glossary is in Appendix A.

Acknowledgements

This book would not exist without the FER *Prevodenje programskih jezika* course and the FRISC architecture documented by [9]; Robert Nystrom’s *Crafting Interpreters* [67], whose voice and pedagogy this book deliberately picks up where the bytecode VM ends; and the classical compilers literature [2, 18, 62, 69] that the appendices lean on so the chapters do not have to.

On a more personal note, this project is older than the book. Its first incarnation was a student team in that same *Prevodenje programskih jezika* course, and I owe my best friends more than a preface can hold. Nikola Sekulić implemented the largest part of that original compiler; Kristijan Štruml and Andrija Krilić were core members of the team and contributed actively throughout; and Boris Komunjer — who studied electrical engineering rather than computing — was a great help in making sense of the machine underneath, and patiently answered more of my questions about computer architecture than either of us could count. To all four: thank you for putting up with me, and for knowing exactly how many years I have been saying that one day I would build my own compiler for FRISC and, once and for all, round off this story.

The deepest debt, though, goes further back. I thank my parents, Milovan and Jasminka, who always encouraged me to play, let my imagination run free, and never tired of my endless questions about the world around me. I thank my sister, Andrea, who made our games more inventive and thanks to whom I was never lonely. And I owe a great deal to my grandfather Stjepan, who passed on to me his love of building. I never became a civil engineer, but in virtual worlds I have raised more structures than I can count — and, in the end, a compiler is just one more of them.

Contents

Foreword	vii
Preface	ix
List of Algorithms	xxv
Notation	xxvii
Abbreviations	xxix
I Foundations	1
1 What we're building, and why it's worth a book	3
1.1 The shortest possible program	5
1.2 Step 1: characters become tokens	5
1.3 Step 2: tokens become a tree	7
1.4 Step 3: the tree gets meaning	9
1.5 Step 4: meaning becomes a recipe	12
1.6 Step 5: the recipe gets cooked down	14
1.7 Step 6: IR becomes machine code	15
1.8 Step 7: machine code runs	17
1.9 What the rest of the book does	19
Practice	20
Notes and further reading	20
2 The shape of a compiler	23
2.1 From a string of characters to a stream of cycles	23
2.2 Why the IR sits in the middle	27
2.3 The FRISC, in one breath	29
2.4 Why we are writing it in Java	32
Your turn	36
Practice	37
Project	37
Notes and further reading	38

II	The Front End	41
3	Reading characters: tokenization	43
3.1	Regular expressions, and why they're not regular	44
3.2	From regex to NFA: Thompson's construction	46
3.3	From NFA to DFA: subset construction	49
3.4	Maximal munch, and the operator-disambiguation problem	54
3.5	Comments, strings, and the lexer's modes	60
3.6	What FRISCcc's lexer actually does	63
	Your turn	66
	Practice	66
	Project	68
	Notes and further reading	69
4	Reading structure: parsing	71
4.1	Grammars: the contract between phases	71
4.2	When grammars go wrong: ambiguity	74
4.3	LL or LR? Choosing a strategy	75
4.4	LR(1) items: snapshots of a parse in progress	77
4.5	Closure and goto: building the item sets	79
4.6	ACTION and GOTO: the parser's table	83
4.7	Tracing a parse on $a + b * c$	87
4.8	Conflicts, and what they tell you	90
4.9	The FRISCcc grammar: precedence by stratification	92
4.10	The implementation in FRISCcc	95
4.11	Worked example: parsing the iterative Fibonacci	96
	Your turn	100
	Practice	100
	Project	101
	Notes and further reading	102
5	Meaning, not just shape: semantics	105
5.1	What the parser cannot know	105
5.2	Types as the shape of meaning	107
5.3	Symbol tables and scope	109
5.4	Typing rules as a calculus	112
5.5	Other invariants: loops, returns, and <code>const</code>	117
5.6	Implementation in FRISCcc	120
5.7	Worked example: typing <code>math_fibonacci_iter</code>	123
	Practice	126
	Project	127
	Notes and further reading	128
	Your turn	129

6	Lowering to typed IR	131
6.1	Why not generate FRISC directly?	131
6.2	Three-address code, in five minutes	133
6.3	Our typed IR	134
6.4	Lowering: a recursive descent through meaning	139
6.5	Well-formedness as a written-down property	145
6.6	Worked example: lowering <code>math_fibonacci_iter</code>	149
	Your turn	152
	Practice	152
	Project	153
	Notes and further reading	154
III	The Back End	157
7	Optimizing without breaking things	159
7.1	What we mean by “optimization”	160
7.2	Constants and arithmetic	162
7.3	Tidying the control-flow graph	164
7.4	Following values around	167
7.5	Loops	173
7.6	Cleanup	175
7.7	The pipeline	178
7.8	Where the proofs live	181
7.9	Worked example: <code>real_prime_sieve</code>	181
	Practice	185
	Project	186
	Your turn	187
	Notes and further reading	187
8	Producing FRISC	189
8.1	The target, up close	190
8.2	From IR temporaries to stack slots	192
	8.2.1 Sizing the frame	193
	8.2.2 What we gave up, and what we would do instead	194
8.3	Instruction selection	196
	8.3.1 The binary-operation pattern	196
	8.3.2 Multiplication without a multiplier	197
	8.3.3 Building large constants	198
8.4	Comparisons and control flow	199
	8.4.1 Branches, jumps, and labels	200
8.5	Memory: loads, stores, and addresses	201
	8.5.1 Array indexing and bounds checks	201
8.6	The calling convention	203
	8.6.1 Prologue and epilogue	204

8.6.2	Passing arguments	205
8.7	Globals and the data section	205
8.8	A second look: the peephole pass	206
8.9	Worked example: <code>real_prime_sieve</code> , all the way down	207
	Practice	208
	Project	210
	Your turn	211
	Notes and further reading	211
9	Runtime: helpers, traps, and frame layout	213
9.1	What a runtime is	214
9.1.1	Pay only for what you use	215
9.2	The call handshake	216
9.2.1	The linkage region	216
9.2.2	Prologue and epilogue	217
9.2.3	The handshake, end to end	217
9.3	Arithmetic the machine refuses	220
9.3.1	Multiplication by shift and add	220
9.3.2	Division and modulo, bit by bit	220
9.4	Fractions without an FPU	223
9.4.1	Conversions are shifts	224
9.4.2	Fixed-point multiply and divide	225
9.5	Traps and the unhappy path	226
9.6	The runtime of <code>real_prime_sieve</code>	228
	Practice	228
	Project	229
	Your turn	230
	Notes and further reading	231
10	Running it: the simulator	233
10.1	Why a simulator, not silicon	234
10.2	The fetch–decode–execute cycle	235
10.3	Inside FRISCjs	236
10.4	Decoding an instruction	238
10.5	Memory, addresses, and how a program stops	240
10.6	Driving the simulator from the compiler	242
10.7	Reading a cycle trace	244
10.8	Counting cycles, counting instructions	245
10.9	Worked example: <code>real_prime_sieve</code> under the simulator	248
	Practice	248
	Project	249
	Notes and further reading	250

IV Beyond Compilation 253

11 Interpreting instead of compiling 255

11.1 Two ways to make a program run	256
11.2 Walking the IR, not the tree	257
11.3 An abstract machine: frames and byte-addressable memory	259
11.4 The interpreter loop	261
11.5 Calls, returns, and the host's call stack	265
11.6 Trapping what FRISC traps	266
11.7 How fast is fast enough?	267
11.8 Watching the anchor run	269
Practice	270
Project	273
Notes and further reading	274

12 A machine of our own 275

12.1 One more step down the spectrum	276
12.2 Designing the bytecode	277
12.3 Lowering the IR to bytecode	279
12.4 The dispatcher loop	282
12.5 Two stacks of our own	285
12.6 Faithful to the same machine	288
12.7 What the extra step buys	288
12.8 Worked example: the sieve as bytecode	290
Practice	293
Project	294
Your turn	295
Notes and further reading	295

V Putting It Together 297

13 Case studies 299

13.1 Recursive Fibonacci: where the call stack lives	300
13.2 GCD and LCM: a helper-dominated cycle budget	304
13.3 Knapsack: iteration and the price of a bounds check	308
13.4 Horner in fixed point: a multiply in every iteration	311
13.5 Kinematics: a program that vanishes	314
13.6 What the three pipelines told us	316
Practice	321
Project	322
Your turn	323
Notes and further reading	323

14 Performance	325
14.1 What to measure, and what we are not measuring	325
14.2 What -O1 buys	327
14.3 Per-pass ablation: which rewrites earn their place	330
14.4 The helper-cycle budget	333
14.5 Convergence: how many sweeps until the IR stabilises	335
14.6 Reproducing these numbers	337
Practice	337
Project	338
Your turn	339
Notes and further reading	339
15 Where to take it next	341
15.1 The whole machine	341
15.2 An honest list of what we left out	344
15.3 A register machine of our own	346
15.4 SSA, and what it would change about Chapter 7	348
15.5 What a JIT would change	350
15.6 A different target, a different source	352
Practice	353
15.7 Books and papers worth your time	355
One last word	356
A A Note on Sources	357
A.1 Lineage	357
A.2 Croatian–English non-terminal mapping	357
A.3 Reading the original materials	358
B FRISC ISA Reference	361
B.1 The processor at a glance	361
B.2 Register file	361
B.3 Instruction encoding	364
B.4 Instruction reference, by mnemonic	365
B.4.1 Data movement	366
B.4.2 Arithmetic and logic	366
B.4.3 Shifts and rotates	366
B.4.4 Memory access	366
B.4.5 Control flow	367
B.5 Condition codes and flags	367
B.6 Memory model	369
B.7 Assembler syntax and directives	369
B.8 The runtime helper set	371
B.9 The calling convention	372

C	Simulator Reference	373
C.1	Where to find it	373
C.2	How FRISCCc drives the simulator	373
C.3	Reading the output	374
C.4	Memory-mapped input and output	375
D	Language and IR Grammar	377
D.1	The specification files	377
D.2	Lexical structure	377
D.3	The accepted C subset	380
D.4	Syntax	380
D.5	Semantics	381
D.6	The source-language specification, in full	381
D.6.1	The lexer specification	382
D.6.2	The parser specification	387
D.6.3	The semantics specification	391
D.7	The intermediate representation	394
D.7.1	Lexical structure	394
D.7.2	Module structure	395
D.7.3	Functions and basic blocks	395
D.7.4	Instruction grammar	395
D.7.5	Type grammar	396
D.7.6	Well-formedness conditions	396
D.8	The IR specification, in full	397
E	Build and Usage Guide	405
E.1	Prerequisites	405
E.2	Building the compiler	405
E.3	Driving the compiler	406
E.4	Running IR directly: the interpreter and the VM	407
E.5	Building and testing the companion	407
F	Solutions	409
F.1	Chapter 1: What we're building, and why it's worth a book	409
F.2	Chapter 2: The shape of a compiler	411
F.3	Chapter 3: Reading characters: tokenization	413
F.4	Chapter 4: Reading structure: parsing	417
F.5	Chapter 5: Meaning, not just shape	422
F.6	Chapter 6: Lowering to typed IR	427
F.7	Chapter 7: Optimizing without breaking things	432
F.8	Chapter 8: Producing FRISC	435
F.9	Chapter 9: Runtime: helpers, traps, and frame layout	439
F.10	Chapter 10: Running it: the simulator	442
F.11	Chapter 11: Interpreting instead of compiling	444
F.12	Chapter 12: A machine of our own	447

F.13 Chapter 13: Case studies	450
F.14 Chapter 14: Performance	453
G Formal Proofs	459
G.1 What “semantic preservation” means here	459
G.2 Determinism of LR(1) parsing	467
G.3 Pass: typed constant folding	470
G.4 Pass: int32 algebraic simplification	473
G.5 Pass: int32 shift simplification	476
G.6 Pass: cast simplification	478
G.7 Pass: ControlFlowSimplificationPass	482
G.8 Pass: UnreachableBlockEliminationPass	489
G.9 Pass: global value propagation	493
G.10 Pass: copy propagation	498
G.11 Pass: Dead Temp Elimination	502
G.12 Pass: Dead Slot Store Elimination	507
G.13 Pass: Load Forwarding	512
G.14 Loop-invariant code motion	516
G.15 Induction-variable strength reduction	520
G.16 Common-subexpression elimination	524
G.17 Tiny-function inlining	528
G.18 Pass: value-range simplification	533
Bibliography	539
Index	545
About the Author	551

List of Figures

1.1	The seven-phase FRISCcc compiler pipeline.	4
1.2	Program states across the seven phases.	18
2.1	The FRISC processor in one block diagram.	30
2.2	The Maven module dependency graph.	34
3.1	Thompson's NFA gadgets for the regex operators.	48
3.2	NFA and DFA for the keyword/identifier conflict <code>int intern</code>	52
3.3	A maximal-munch trace through the loop header of <code>math_fibonacci_iter</code>	57
4.1	Two parse trees for the same token sequence.	72
4.2	The dot position inside an LR item is the parser's reading head.	78
4.3	A fragment of the LR(1) automaton for the three-rule expression grammar.	82
4.4	Parse-stack animation for <code>a + b * c</code>	89
4.5	The precedence cascade for <code>a + b</code>	94
4.6	Top-level skeleton of the parse tree for the anchor program.	97
5.1	The scope stack while typing <code>t = a + b</code>	110
5.2	A typing derivation for <code>a + b</code> in the loop body.	113
5.3	Post-order visit pattern over a fragment of the loop's parse tree.	122
6.1	Frame layout of the Fibonacci anchor.	136
6.2	Lowering a for-loop to four blocks.	145
6.3	The verifier catches a definition-before-use violation.	147
7.1	CFG of a loop, before and after.	166
7.2	The optimiser pipeline.	179
7.3	Per-pass IR-line-count reduction across the portfolio.	180
8.1	The FRISC code generator's internal pipeline.	191
8.2	The activation record FRISCcc builds for <code>main</code>	195
8.3	Comparison materialisation as a four-block diamond.	199
8.4	Lowering an array subscript with bounds checks.	202
9.1	On-demand emission of runtime helpers.	215
9.2	The linkage region between two frames.	218
9.3	The call handshake for <code>t = p * p</code>	219
9.4	The Q16.16 fixed-point format.	224

10.1	How FRISCcc drives the FRISCjs simulator.	237
10.2	Decoding an ALU-format FRISC instruction word.	240
10.3	The dynamic instruction mix of <code>real_prime_sieve</code>	246
11.1	The interpreter’s dispatch cascade.	258
11.2	The interpreter’s memory model.	260
11.3	The control-flow graph the interpreter walks.	263
11.4	The cost of interpretation, in steps.	268
12.1	How one IR instruction becomes bytecode.	277
12.2	The anatomy of the bytecode VM.	284
12.3	The VM’s own call stack, mid-recursion.	287
12.4	Interpreter steps versus bytecode dispatches.	289
13.1	The recursion call tree.	301
13.2	Three back ends, five programs.	318
13.3	Where native cost comes from.	319
13.4	The helper budget.	320
14.1	What <code>-O1</code> buys, dynamic versus static.	328
14.2	Per-pass dynamic ablation across the suite.	332
14.3	Helper-cycle share per program at <code>-O1</code>	334
14.4	Convergence of the <code>-O1</code> fixpoint loop.	336
15.1	The whole FRISCcc system on one page.	342
15.2	The extension horizon, grouped by where it lands.	345
15.3	Stack bytecode versus register bytecode for one IR line.	347
15.4	SSA form and the phi-node.	349
15.5	Tiering: how a JIT sits on top of the bytecode VM.	351
B.1	The FRISC fetch–decode–execute cycle.	362
B.2	The FRISC register file and its conventional roles.	363
B.3	The 32-bit FRISC instruction word (ALU format).	364
B.4	The FRISC address space.	370
G.1	Small-step rules for the representative IR forms.	462

List of Tables

4.1	The complete ACTION/GOTO table for the three-rule expression grammar.	86
4.2	Shift-reduce trace for the input <code>a + b * c</code> .	89
5.1	Implicit promotions and explicit casts in the FRISCcc C subset.	117
6.1	The right-hand side opcodes of FRISCcc's IR.	138
6.2	The IR's type catalogue.	138
8.1	The FRISCcc register conventions.	192
8.2	Instruction selection: IR operations to FRISC.	198
10.1	The first twelve cycles of <code>real_prime_sieve</code> .	245
10.2	<code>real_prime_sieve</code> at -O0 versus -O1, in cycles.	247
13.1	The full three-way comparison.	317
14.1	The fourteen-program performance suite.	327
14.2	Per-program -O0 versus -O1.	329
14.3	Per-pass dynamic ablation, the active eight.	332
B.1	The FRISC opcode map.	365
B.2	Status-register bits.	368
B.3	Jump conditions.	368
B.4	The FRISCcc runtime helper routines.	371
D.1	The front-end specification files.	378
D.2	The four scanner states.	378
D.3	The token alphabet.	378
D.4	The accepted C subset at a glance.	380
E.1	Build prerequisites.	405
E.2	The compiler phase flags.	406
E.3	The companion modules.	408

List of Algorithms

2.1	FRISCCc pipeline driver.	26
3.1	Thompson construction of an NFA from a regex AST.	47
3.2	Epsilon-closure of an NFA state set.	51
3.3	Subset construction of a DFA from an NFA.	51
3.4	Maximal-munch token scanner.	58
4.1	LR(1) item-set closure.	80
4.2	LR(1) GOTO transition.	81
4.3	Canonical collection of LR(1) item sets.	83
4.4	ACTION/GOTO table construction.	84
4.5	Shift–reduce LR(1) parsing driver.	88
5.1	Lexical-scope stack operations.	111
5.2	Expression type-checker.	116
5.3	Statement type-checker.	119
5.4	Post-order visitor over the parse tree.	121
6.1	Expression lowering to a temporary (r-value mode). Each case emits at most a constant number of IR instructions and recurses on direct children; the total work is linear in the size of the source expression.	141
6.2	Expression lowering to a pointer (l-value mode). The four cases are exactly the four kinds of expression that can appear on the left of an assignment in our source language; any other expression kind invoked in l-value mode is a semantic-analysis bug that should have been caught earlier.	142
6.3	Statement lowering with control flow. Loops follow the four-block pattern (init, condition, body, step), with the step block elided for while. Fresh labels are allocated at every loop and conditional boundary; break and continue resolve against the innermost enclosing loop via the label stack.	144

7.1	Typed constant folding.	163
7.2	Worklist dataflow analysis.	169
7.3	Loop-invariant code motion.	174
8.1	Frame sizing and temporary slot assignment.	194
8.2	The general binary-operation lowering pattern.	197
8.3	Materialising a comparison into a Boolean value.	200
8.4	Lowering an array subscript.	203
8.5	Function prologue and epilogue.	204
8.6	The FRISC peephole optimiser.	207
9.1	Function prologue and epilogue.	219
9.2	Signed integer multiplication, <code>F_MUL</code>	222
9.3	Restoring division (core loop).	223
9.4	Fixed-point multiplication, <code>F_FMUL</code>	226
9.5	Array bounds checking.	227
10.1	The fetch–decode–execute cycle.	235
10.2	Instruction decode by bit-field extraction.	239
10.3	The simulator-driver loop with a watchdog.	243
11.1	The block-walking interpreter loop.	262
11.2	Evaluating a right-hand side.	264
11.3	The call protocol.	265
12.1	Lowering a right-hand side to stack code.	279
12.2	The bytecode dispatch loop.	283
12.3	The call protocol on an explicit stack.	285
14.1	Per-pass dynamic ablation.	331

Notation

One short table, used throughout. Anything beyond this lives in [Appendix B](#), [Appendix D](#), or the back-of-book index.

Symbol	Meaning	Chapter
$\mathcal{S}, \mathcal{T}$	source language, target ISA	1
$\Gamma, \Gamma \vdash e : \tau$	typing context, judgement	5
$\llbracket \cdot \rrbracket$	semantic-evaluation brackets	2
σ	machine state (memory + registers)	Appendix G
$\rightsquigarrow, \rightsquigarrow^*$	single step, reflexive-transitive closure	Appendix G
$\varepsilon\text{-cl}$	ε -closure	3
$t0, t1, \dots$	IR temporaries	6
$R0\text{--}R7$	FRISC general-purpose registers	8
FP, SP	frame and stack pointers	8
$L0, L1, \dots$	basic-block labels	6
$F_MUL, F_DIV, \dots$	runtime helper routines	9

Abbreviations

The acronyms below are used throughout the book. Symbols (as opposed to abbreviations) are collected in the [Notation](#) table; FRISC opcodes and register names are catalogued in [Appendix B](#). The two predictive grammar sets FIRST and FOLLOW ([Chapter 4](#)) and the two LR parse-table components ACTION and GOTO are spelled in small capitals wherever they appear and are not repeated here.

Abbreviation & meaning		Abbreviation & meaning	
ABI	application binary interface	JIT	just-in-time (compilation)
ALU	arithmetic–logic unit	JVM	Java virtual machine
ANTLR	ANother Tool for Language Recognition	LALR	look-ahead LR (parsing)
AST	abstract syntax tree	LICM	loop-invariant code motion
CFG	control-flow graph	LLVM	the LLVM compiler infrastructure
CRC	cyclic redundancy check	LR	left-to-right, rightmost (parsing)
CSE	common-subexpression elimination	MIPS	MIPS instruction-set architecture
DFA	deterministic finite automaton	NFA	nondeterministic finite automaton
FER	Faculty of Elec. Eng. & Computing	PC	program counter
FP	frame pointer (register R5)	PLT	Prog. Language Translation
GCC	GNU Compiler Collection	Q16.16	16.16 fixed-point format
GLR	generalised LR (parsing)	RISC	reduced instruction-set computer
GVP	global value propagation	SLR	simple LR (parsing)
IEEE	as in the IEEE 754 standard	SP	stack pointer (register R7)
IR	intermediate representation	SR	status register
ISA	instruction-set architecture	SSA	static single assignment
ISO	Int’l Organization for Standardization	TCF	typed constant folding
IV	induction variable	VM	virtual machine

The optimisation levels -O0 (no optimisation) and -O1 (the baseline pipeline) name the two configurations measured in [Chapter 14](#). FRISC abbreviates *FER Reduced Instruction-Set Computer*, the educational architecture this book targets; FER is the *Fakultet elektrotehnike i računarstva* of the University of Zagreb.

Part I

Foundations

Chapter 1

What we're building, and why it's worth a book

“A programming language is low level when its programs require attention to the irrelevant.”
— Alan Perlis, *Epigrams on Programming*, 1982

Most of us have written a C program. We typed some text into a file, ran `gcc` or `clang` on it, and a moment later there was an executable in the working directory that did roughly what we asked it to. Between the typing and the running, something extraordinary happened, and almost none of us could give a faithful account of what it was. The program was characters; then it wasn't. The program was a tree; then it was instructions. The compiler ate one shape of thing and produced another, and the bit in the middle was a black box we had been trained to trust.

This book is about prying open that box. Not by reading a paper that describes a compiler, and not by skimming the architecture chapter of a textbook, but by walking through every phase of a real, working, modest compiler that we built from scratch. The compiler is called `FRISCcc`. It accepts a deterministic subset of C and emits assembly for an educational RISC machine called the FRISC. Both the subset and the target are small enough to understand completely and large enough to be interesting. Over the course of fifteen chapters we are going to follow programs from the moment they are a string of bytes on disk until the moment a simulated processor executes the last instruction and prints an answer.

To start, we will take the smallest program that actually exercises every phase of the compiler, and we will trace it from one end of the pipeline to the other ([Figure 1.1](#) shows the whole route at a glance). Every phase named, none explained in depth. The point of this chapter is not to teach lexing or parsing or code generation; later chapters do that, one at a time, in detail. The point is to show you the shape of the journey, so that by the end of these twenty-odd pages you know what each phase is for, what its output looks like, and roughly why it earns its place in the pipeline. Then we can spend the rest of the book opening each box and finding out how it works.

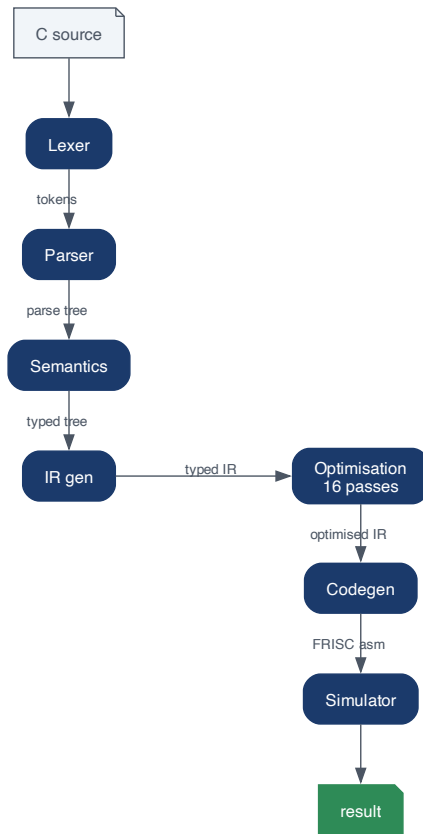


Figure 1.1: The seven phases of the FRISCCc compiler, read top-to-bottom on the left, then top-to-bottom on the right. Each arrow is labelled with the artifact handed from one phase to the next: tokens, parse tree, typed tree, IR, optimised IR, FRISC assembly. The dog-eared boxes are boundary artifacts (the input source file and the simulator's printed result); the rounded navy boxes are the seven compiler phases that the rest of the book opens up one by one.

```
1 // EXPECT: 15
2 // The shortest possible program that exercises the compiler's full pipeline.
3 // Anchor example for Chapter 1 of the book.
4
5 int main(void) {
6     return 5 * 3;
7 }
```

Listing 1.1: The anchor program. Seven lines: three comments, one blank, and three lines of actual code. The function `main` takes no arguments and returns the product of five and three: fifteen.

1.1 The shortest possible program

Here is a complete C program that we will follow from cover to cover of this chapter.

It is hard to write a smaller C program that still does something. We have a function definition, a return statement, a binary operator, two integer literals, and a semicolon. The expected output is fifteen — the comment on line one says so, and arithmetic confirms it. A human reader could predict the answer without taking a breath.

The compiler, however, cannot. The compiler does not understand the program; it transforms it. And the transformation is not a single step. The compiler reads [Listing 1.1](#) as a sequence of bytes, and in seven distinct, identifiable phases it converts those bytes into FRISC machine code that, when run on a simulator, prints 15. Each phase takes an artifact produced by the previous phase, restructures it according to a particular kind of knowledge, and hands it to the next. The artifacts get progressively closer to something a machine can execute, and progressively further from something a human wants to read.

Let us watch each phase happen.

1.2 Step 1: characters become tokens

Before the compiler can do anything interesting with the program, it has to chop the text into atoms. The smallest meaningful unit of a programming language is not a character; it is a *token*, the lexical equivalent of a word. The keyword `return` is a token. The integer literal `5` is a token. The asterisk is a token. The semicolon is a token. Whitespace and comments, by contrast, are not tokens at all: they exist for the benefit of humans and disappear at the lexer's boundary.

The phase that does the chopping is called the *lexer* (or sometimes the scanner). If we hand the lexer the seven-line file from [Listing 1.1](#), it produces the following list of tokens.

```

1  LEXER OUTPUT
2  =====
3
4  Token Table
5  -----
6  Index      Token Name      Token Value
7  0          KR_INT       int
8  1          IDN         main
9  2          L_ZAGRADA    (
10 3          KR_VOID      void
11 4          D_ZAGRADA    )
12 5          L_VIT_ZAGRADA {
13 6          KR_RETURN    return
14 7          BROJ         5
15 8          ASTERISK     *
16 9          BROJ         3
17 10         TOCKAZAREZ   ;
18 11         D_VIT_ZAGRADA }
19
20 Uniform Token Stream
21 -----
22 Token Name      Source Row  Token Table Index
23 KR_INT          5          0
24 IDN             5          1
25 L_ZAGRADA       5          2
26 KR_VOID         5          3
27 D_ZAGRADA       5          4
28 L_VIT_ZAGRADA   5          5
29 KR_RETURN       6          6
30 BROJ            6          7
31 ASTERISK        6          8
32 BROJ            6          9
33 TOCKAZAREZ      6          10
34 D_VIT_ZAGRADA   7          11

```

Listing 1.2: The lexer's complete output for the anchor program. The top table assigns each distinct token a value and an index; the bottom table is the linear stream the parser will consume, annotated with the source line on which each token appeared. The names are in Croatian because the canonical PLT-FER specification on which FRISCcc is based uses Croatian non-terminals; KR_INT reads as “keyword *int*”, IDN as “identifier”, L_ZAGRADA as “left parenthesis”, TOCKAZAREZ as “semicolon”. The full glossary lives in Appendix A.

Twelve tokens. That is everything the rest of the compiler needs to know about the surface text. Once the lexer is done, the bytes of the source file have served their purpose; nothing downstream ever looks at [Listing 1.1](#) again. The tokens carry the line number for the sake of error messages, but otherwise the conversion is total: from here forward, the program *is* the token stream.

The lexer’s job is mechanical but not trivial. It has to recognise that `int` is a keyword but `intern` would be an identifier; that `5` is one integer literal but `53` is another; that `*` is a single asterisk but `*=` would be a different token entirely. The standard way to do this — and the way FRISCC does it — is to compile a regular expression for each token kind into a finite-state machine and run all the machines in parallel against the input, picking the longest match at every step. Chapter 3 will show that machinery in detail and prove why it works.

The class of languages that lexers can describe is exactly the class of *regular languages*: those recognisable by a finite-state machine with no extra memory. This is the bottom rung of the Chomsky hierarchy. Regular languages are weak enough that “balanced parentheses” is beyond them — you cannot write a regular expression that accepts `((x))` but rejects `((x)))`. That weakness is also their strength: lexers run in linear time with constant memory per token, no surprises. Anything more powerful belongs to the parser, which is the next phase.

So we have twelve tokens. They are atoms, and like atoms they sit there inert. They do not know that `*` is supposed to have two operands. They do not know that `return` is supposed to be followed by an expression. They have no idea what the shape of the program is. Giving them shape is somebody else’s job.

1.3 Step 2: tokens become a tree

The shape of the program is a tree. Every binary operator is the parent of its two operands. Every function definition is the parent of its signature and its body. Every statement is a leaf or has children, in a hierarchy that ultimately roots in a single node representing the whole translation unit. The phase that builds this tree is the *parser*.

The parser’s input is the token stream we just saw; its output is a *parse tree* (sometimes called a concrete syntax tree, since it faithfully records every grammatical step). For our seven-line program, the parse tree comes out 46 lines long. We will look at the part that covers the body of `main`.

Two things about [Listing 1.3](#) are worth pointing out. First, multiplication has won out over addition, comparison, logical-and, and all the other binary operators with looser precedence: the `<multiplikativni_izraz>` sits at the bottom of the precedence ladder, closest to the literals, which is why `5 * 3` is grouped correctly without any parentheses in the source. The parser made that choice on our behalf, by following a grammar that encodes C’s precedence

```

19      <naredba>
20      <naredba_skoka>
21      KR_RETURN , return
22      <izraz>
23      <izraz_pridruzivanja>
24      <log_ili_izraz>
25      <log_i_izraz>
26      <bin_ili_izraz>
27      <bin_xili_izraz>
28      <bin_i_izraz>
29      <jednakosni_izraz>
30      <odnosni_izraz>
31      <aditivni_izraz>
32      <multiplikativni_izraz>
33      <multiplikativni_izraz>
34      <cast_izraz>
35      <unarni_izraz>
36      <postfiks_izraz>
37      <primarni_izraz>
38      BROJ , 5
39      ASTERISK , *
40      <cast_izraz>
41      <unarni_izraz>
42      <postfiks_izraz>
43      <primarni_izraz>
44      BROJ , 3
45      TOCKAZAREZ , ;
46      D_VIT_ZAGRADA , }

```

Listing 1.3: An excerpt from the parse tree, covering the body of main. Each indented line is a node; non-terminals appear in angle brackets, terminals on their own. The full tree is in `listings/return_5x3/ast.txt`. Notice the deep chain from `<izraz>` (expression) down through `<multiplikativni_izraz>` (multiplicative expression) and finally to the literal 5: every precedence level of C has its own non-terminal, and the parser must descend through all of them to reach the leaves.

rules directly into its non-terminal structure. We will spend most of [Chapter 4](#) on how that grammar is designed and why it works.

Second, the tree is enormous compared to the source. Seven lines of C have become forty-six lines of tree, with most of the depth devoted to *walking through* precedence levels we never actually used. This is the price of a strictly literal parse tree, and it is the reason real compilers eventually flatten it into an *abstract* syntax tree where the noise is collapsed away. FRISCcc carries the verbose tree all the way through semantic analysis, because the same structure is convenient for the next phase. The verbosity is a teaching choice, not an optimisation one.

The parser is doing something a regular expression can never do: it is tracking nested structure. Concretely, it pushes onto a stack each time it opens a new construct and pops when the construct closes. The class of languages that can be described this way is called *context-free*, and the formalism is the *context-free grammar* of Backus-Naur-form fame [64]. FRISCcc uses a particular kind of parser called *LR(1)*, which is deterministic, linear-time, and can handle the precedence-and-associativity rules of C with no backtracking. The classical reference for these parsers is Knuth's 1965 paper [48].

The tree knows the structure of the program. What it does not know is whether the program is *meaningful*: whether the operands of `*` are types that can actually be multiplied, whether `main` returns the right kind of thing, whether the names mentioned have been declared. Discovering all that is the next phase's job, and it is one of the most subtle parts of the compiler.

1.4 Step 3: the tree gets meaning

A parse tree describes what the program *says*. It does not describe what the program *means*. The phase that bridges the gap is called *semantic analysis*, and it is the most underrated part of any compiler. Without it, you can write `int x; x();` and the parser will smile politely.

Semantic analysis walks the parse tree and decorates every node with the information needed to verify that the program is legal and to translate it. For each variable use, it finds the declaration; for each expression, it infers a type; for each function call, it checks that the arguments match the signature; for each statement, it tracks whether the surrounding function is guaranteed to return. It also builds the *symbol table*, which is the mapping from names to declarations that the rest of the compiler will use for the rest of its life.

For our anchor program almost nothing controversial happens. The literal `5` has type `int`. The literal `3` has type `int`. Multiplying two `ints` yields an `int`. The function `main` is declared to return `int`. The return statement returns an expression of type `int`. Everything checks out, the semantic tree is well-formed, and no diagnostics are emitted.

```

1  SEMANTIC TREE REPORT
2  =====
3
4  Symbol Table
5  -----
6  Summary:
7  - total scopes: 2
8  - total symbols: 1
9  - variables: 0
10 - constants: 0
11 - functions: 1
12
13 Scopes and symbols:
14
15 Scope #1 (depth=0, parent=none)
16 Name                Kind      Type / Signature      Details
17 main                function  int ()                defined=true,
    ↪ return=int, params=0
18
19 Scope #2 (depth=1, parent=#1)
20 Name                Kind      Type / Signature      Details
21 (no symbols)        -        -                    -
22

```

Listing 1.4: The symbol-table portion of the semantic-analysis report. We have two scopes (the global scope and the function's own), one symbol (`main`), and zero variables — the inner scope is empty because the function has no locals and `void` parameter lists declare nothing. The full report continues with the parse tree re-annotated with type, l-value, and constant-value flags on every node; an excerpt of that annotation is in [Listing 1.5](#).


```

51      <izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
52      <izraz_pridruzanja> [type=int, lvalue=false, constValue=false, containsReturn=false]
53      <log_ili_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
54      <log_i_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
55      <bin_ili_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
56      <bin_xili_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
57      <bin_i_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
58      <jednakosni_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
59      <odnosni_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
60      <aditivni_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
61      <multiplikativni_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
62      <multiplikativni_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
63      <cast_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
64      <unarni_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
65      <postfiks_izraz> [type=int, lvalue=false, constValue=false,
66      ↪ containsReturn=false]
67      <primarni_izraz> [type=int, lvalue=false, constValue=false,
68      ↪ containsReturn=false]
69      BROJ (line=6, lexeme=5)
70      <cast_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
71      <unarni_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
72      <postfiks_izraz> [type=int, lvalue=false, constValue=false, containsReturn=false]
73      <primarni_izraz> [type=int, lvalue=false, constValue=false,
74      ↪ containsReturn=false]
75      BROJ (line=6, lexeme=3)
76
77      TOCKAZAREZ (line=6, lexeme=;)
78      D_VIT_ZAGRADA (line=7, lexeme=})

```

Listing 1.5: The same parse tree from Listing 1.3, now with each node annotated with its inferred type and l-value status. The expression $5 * 3$ carries `type=int` on every level; the multiplication itself is correctly identified as not an l-value (you cannot assign to it); and the surrounding `<izraz>` is also typed `int`, which is what the return statement requires. Every check that the compiler does — type agreement, value-category compatibility, declared-before-used — happens in this pass, on this tree.

But the silence is doing real work. Had we written `return 5 * "hi";` the parser would have happily built a tree — it has no opinion about types — but semantic analysis would have raised an error during type inference, because there is no rule that says “`int * char[3]` is an `int`”. Most of what makes a programming language *strict* lives in this phase. C is permissive about some things (it will let you multiply an `int` and a `char` without complaining, and silently promote one) but draws a line at others (multiplying an `int` and a string literal is forbidden). The set of rules that draws the line is what people mean when they say “the type system”.

Why is type checking a separate phase from parsing? It does not have to be — there are real-world languages where the two are interleaved (early Pascal compilers, some Scheme implementations) — but the separation is almost always cleaner, because the grammar and the type system are governed by entirely different kinds of rules. The grammar says how programs are *shaped*; the type system says when programs are *coherent*. Mixing them tends to produce parsers that are both hard to extend and hard to reason about. Modern compilers, FRISCCc included, run the parser first and the type checker over the parser's output.

The formal theory of how types “flow” through a program is one of the most active areas of programming-language research. Two of the most durable textbooks on the subject are Pierce's *Types and Programming Languages* [69] and Harper's *Practical Foundations for Programming Languages* [34]. Chapter 5 will draw on both.

Now we know what the program means. We know that `main` is a function returning `int`, that its body computes the integer fifteen, and that the type of every sub-expression is consistent. What we do not yet know is how to translate any of that into instructions for a machine. The gap is enormous: a parse tree is a hierarchical, abstract object; a machine wants a flat, concrete sequence of operations on registers. We have to lower one to the other, and the move is so big that we are going to do it through a deliberate intermediate stop.

1.5 Step 4: meaning becomes a recipe

If we tried to translate the semantic tree directly into FRISC assembly, we would have to write a translator that simultaneously knows the C language, the FRISC instruction set, the FRISC calling convention, every addressing mode, and the entire machinery of register allocation. It is not impossible — early compilers worked roughly this way — but it is unmaintainable. Worse, it commits us to a single target. If we later decided to retarget the compiler to a different machine, we would have to rewrite the translator from scratch.

The standard escape from this trap is to introduce an *intermediate representation*: a small, regular language sitting between the high-level source and the low-level target. The front end's job is to translate source into IR. The back end's job is to translate IR into target code. The two ends are now decoupled, and each is simpler than the monolithic translator would have been. FRISCCc's IR looks like this.

```
1  .program
2
3  .func main():int32
4      .frame locals=0 bytes align=4
5      .slots
6      .blocks
7      L0:
8          t0 = mul #5:int32, #3:int32 : int32
9          ret t0
10     .endfunc
11
12 .endprogram
```

Listing 1.6: The unoptimised intermediate representation. `.func main():int32` declares a function returning a 32-bit integer; the `locals=0` field of the `.frame` directive says it has no local-variable storage; the basic block `L0` contains two instructions. The first multiplies the constants `#5:int32` and `#3:int32`, storing the result in a temporary called `t0`. The second returns that temporary. The full IR grammar is documented in Appendix D.

Listing 1.6 is what the front end produces and the back end consumes. Notice the shape: it is a list of instructions, each operating on at most one destination and a couple of operands. This style is called *three-address code*, and it has been the dominant IR shape since the 1970s. Each instruction is small enough to map to one or two machine instructions; each is large enough to express any source-language operation. The IR has temporaries (`t0` here, and later `t1`, `t2`, and so on), labels (`L0` here, and others for control flow), and typed constants (`#5:int32`). It does not have loops, parentheses, declarations, or any of the syntactic furniture of the source. The verticality of the tree has become the linearity of a recipe.

The aphorism attributed to David Wheeler — “all problems in computer science can be solved by another level of indirection” — is nowhere more visible than at the IR. By inserting one extra representation between the front and back ends, we turn an $M \times N$ engineering problem (M source languages times N target machines) into an $M + N$ problem (M front ends and N back ends sharing one IR). This is the architectural insight behind GCC, behind LLVM, and behind FRISCCc. The cost is the IR itself, which has to be designed carefully enough that translations into it are natural for every front end and translations out of it are natural for every back end. LLVM in particular [53] has pushed this idea to remarkable lengths; we will see in Chapter 6 how the design trade-offs look for a much smaller language.

Look at Listing 1.6 again, and read the multiplication line charitably. We are asking the

compiler to multiply 5 by 3 at run time. Every time the program is invoked, the simulated processor will load five into a register, load three into a register, call a helper that multiplies them, and produce fifteen. We could spare it all that work. The value of $5 * 3$ is fifteen, and we already know it is fifteen, and there is no reason at all to leave the multiplication in the program. The compiler should fold it away.

That folding is the next phase's job.

1.6 Step 5: the recipe gets cooked down

The optimiser is a sequence of *passes*: small transformations that each take an IR program as input, look for a specific pattern, and produce a (hopefully simpler) IR program as output. FRISCCc has sixteen of them. They run in a fixed pipeline, sometimes more than once, until the program stops changing. For our anchor program, the only pass that does useful work is called *typed constant folding*, and it does exactly what the name suggests: when it sees an arithmetic instruction whose operands are all known at compile time, it replaces the instruction with the precomputed result. After the pipeline runs, the IR looks like this.

```

1  .program
2
3  .func main():int32
4    .frame locals=0 bytes align=4
5    .slots
6    .blocks
7    L0:
8      t0 = #15:int32
9      ret t0
10 .endfunc
11
12 .endprogram

```

Listing 1.7: The IR after the optimisation pipeline. Compare to [Listing 1.6](#): the `mul` instruction is gone, replaced by a direct assignment of the constant `#15:int32` to the temporary `t0`. One instruction saved, one helper call saved, one multiplication saved at run time, and the program still computes exactly the same value.

One line shorter. One run-time multiplication eliminated. For a program this small, the gain looks ridiculous; for a real program, where these patterns appear hundreds of times — inside loops, inside formulas, inside macro expansions — the cumulative effect is substantial. [Chapter 14](#) will show concrete measurements for the entire benchmark suite. For now, we just want to notice that the optimiser *can* fold constants, and that doing so

is a transformation: the input is IR, the output is IR, and the semantics of the program (what value it returns when run) is preserved.

Preservation is the whole game. An optimiser that produces a faster program but a different answer is not an optimiser; it is a bug. Each of the sixteen passes in FRISCCc has to be argued to preserve the meaning of the program it transforms, and Chapter 7 will give the argument in detail (with full semantic-preservation proofs for all sixteen passes in [Appendix G](#)). The intuition is straightforward in this case — $5 * 3$ evaluates to 15 under any sensible semantics — but for, say, removing a load whose value is already in a register, or hoisting an expression out of a loop, the argument is much more delicate.

Donald Knuth’s famous 1974 essay [49] contains the line “premature optimization is the root of all evil”, and the line has been weaponised so often that the surrounding caveat has been almost lost. Knuth’s actual point is that *ninety-seven per cent of the time*, optimisation is wasted effort. The remaining three per cent — the critical sections of a program, the inner loops, the constant-folded multiplications repeated billions of times — absolutely deserve the attention. A compiler is, in some sense, automating exactly that three per cent: it finds the small, local, high-leverage transformations that a programmer should not have to do by hand and does them at scale. It is optimisation in Knuth’s blessed sense.

But this is still IR. The optimiser left us with a five-line program in a language no real processor speaks. We still have one more language to translate down to: the actual FRISC assembly.

1.7 Step 6: IR becomes machine code

The code generator takes the optimised IR and emits FRISC assembly. The FRISC is an educational RISC machine developed at the Faculty of Electrical Engineering and Computing (FER), University of Zagreb, where it was the teaching machine of the *Computer Architecture 1* (*Arhitektura računala 1*) course; in the *Programming Language Translation* (PLT) course it is simply the target our compiler emits. We will describe it properly in the next chapter, and exhaustively in Appendix B. For now, all we need to know is that it has eight general-purpose registers (R0–R7, of which R5 is the frame pointer and R7 is the stack pointer), instructions for loading, storing, jumping, arithmetic, and function call/return, and an absolutely unambiguous syntax. Here is what the code generator produces for our program.

We have gone from twelve tokens to forty-six lines of tree to a typed IR to a flat assembly listing in which every line corresponds to roughly one machine instruction. [Listing 1.8](#) is the first artifact in the pipeline that a piece of physical (or simulated) silicon could actually execute. Reading it as a human, the body of `main` is doing more work than it strictly needs to: it stores 15 into a memory slot and immediately loads it back. That redundancy is

```

1  ; =====
2  ; PPJ Compiler - Generated FRISC Assembly Code
3  ; Source: program.c
4  ; Generated: 2026-05-20 22:32:15
5  ; =====
6  ; =====
7  ; Program entry point and initialization
8  ; =====
9      MOVE 40000, R7      ; Initialize stack pointer (SP)
10     CALL F_MAIN        ; Call main
11     HALT               ; Program end
12 ; =====
13 ; Function Definitions
14 ; =====
15 F_MAIN                ; Function: main
16     PUSH R5            ; Save old FP
17     MOVE R7, R5        ; Set FP
18     SUB R7, 8, R7      ; Allocate locals/temps
19     MOVE 1, R1          ; Zero local words
20     MOVE R5, R0        ; Local zero base
21     SUB R0, 4, R0       ; Local zero ptr
22     MOVE 0, R2          ; Zero
23 L_ZERO_2
24     STORE R2, (R0)      ; Clear
25     ADD R0, 4, R0
26     SUB R1, 1, R1
27     JP_NE L_ZERO_2
28 L_MAIN_L0
29     MOVE 0F, R0         ; Store temp t0
30     STORE R0, (R5-8)    ; Store temp t0
31     LOAD R0, (R5-8)     ; Load temp t0
32     MOVE R0, R6         ; Set return value
33 L_EXIT_F_MAIN_1
34     ADD R7, 8, R7      ; Deallocate locals/temps
35     POP R5             ; Restore FP
36     RET                ; Return
37 ; =====
38 ; Global Variables
39 ; =====

```

Listing 1.8: Generated FRISC assembly. The file opens with the program entry point (lines 9–11): initialise the stack pointer to address 40000, call F_MAIN, halt the processor. Lines 15–36 define the function main itself, with the standard prologue (PUSH R5, set the new frame pointer, allocate two words of local storage), the body (load the constant 0F (hexadecimal for fifteen) into R0; store it via the frame pointer; reload it; move it to the return-value register R6), and the epilogue (deallocate, restore the old frame pointer, RET).

partly an artefact of the unoptimised lowering style (the IR temporary `t0` was given a stack slot, and the code generator is faithful about realising it), and partly deliberate — it keeps the codegen logic simple at the cost of a few extra cycles. Chapter 8 will discuss the trade-offs, and [Chapter 14](#) will measure how much they cost.

What is much more interesting than any individual instruction is the *shape* of the function: prologue, body, epilogue. The shape is a contract — the FRISC *calling convention* — and every function in every program ever compiled by FRISCcc obeys it. The caller knows that arguments arrive in particular registers, the return value comes back in `R6`, and the callee will not clobber the frame pointer. The callee knows that the stack will be in a predictable state on entry and that it must restore it on exit. The two sides agree, and the agreement is what makes function calls possible at all.

Calling conventions are best understood as social contracts. Like all social contracts, they exist because both parties benefit from the agreement and would be worse off without it. There is no law of nature that says argument zero goes in `R1`; the C ABI on a different architecture might put it in a totally different place. What matters is that everyone agrees, that the agreement is documented, and that no function is allowed to break it unilaterally. When you call a library function written in assembly, you are trusting the convention as much as you are trusting the function.

The FRISC assembly in [Listing 1.8](#) is text. It has not been executed yet. The last thing we need to do is run it.

1.8 Step 7: machine code runs

The FRISC is an educational architecture, which means it exists more fully in simulation than in silicon. We use a JavaScript implementation called `friscjs` that loads an assembly file, parses it, and executes the instructions one by one until the processor halts. There is no operating system, no scheduler, no virtual memory. There is a flat address space, eight registers, four condition flags, and a fetch–decode–execute loop. When the program halts, the simulator prints whatever `main` returned, on the assumption that we wanted to know.

For our anchor program, the entire simulator output is one line.

1 | 15

Listing 1.9: The complete output produced by running the generated FRISC assembly on the `friscjs` simulator. The program halts, the return value is read from `R6`, and the answer is printed: fifteen.

That number on the page is the same number that a human reader of [Listing 1.1](#) would have written down without any of the machinery we just walked through. Seven phases. Twelve tokens. Forty-six lines of tree. Two scopes and one symbol. Two lines of IR before the optimiser, two lines after. Thirty-nine lines of FRISC assembly. A simulator with a fetch-decode-execute loop. And at the end of all of it, the integer fifteen, the same integer we knew the answer was before any of this started. [Figure 1.2](#) retells the same journey from the artifacts' side: what the program *is* at each phase boundary, rather than which phase does the work.

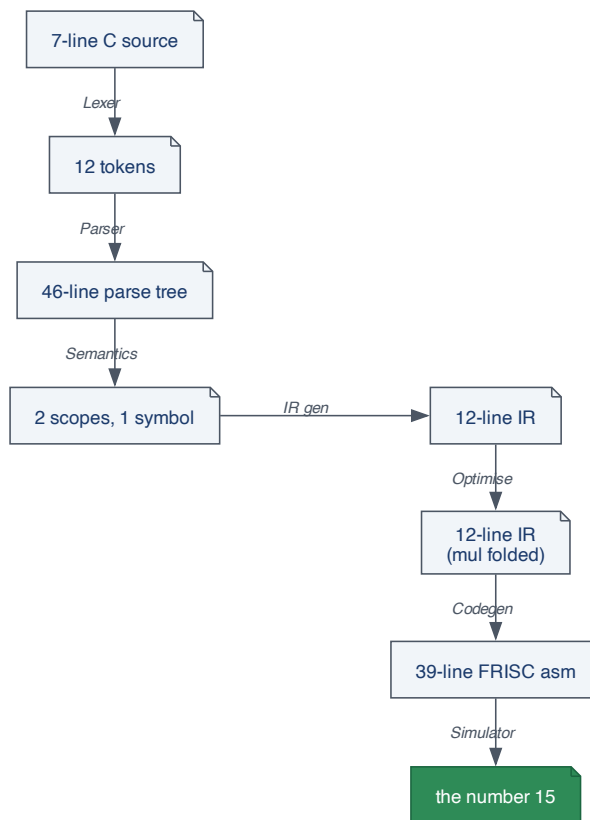


Figure 1.2: The same pipeline as the opening figure, read by what the program *is* at each phase boundary rather than by which phase processes it. The chain begins as seven lines of C and ends as the integer fifteen; every box in between is a checkpoint where the compiler has finished one phase and not yet started the next. Each arrow is labelled with the phase that performs the transformation. Compare to the navy phase-boxes of [Figure 1.1](#): the same seven transformations, told from the artifacts' side.

What was the point? Why go through seven phases to compute something a human could compute in their head? Two reasons. The first is that the program does not have to be this small. The same pipeline, exactly the seven phases, exactly the same chain of artifacts, processes a program that does Eratosthenes' sieve, or iterative Fibonacci, or polynomial evaluation with Horner's rule. We chose a tiny program so we could see every phase on one page; we could have chosen a thousand-line program and the structure would have been identical. The second reason is that this *is* how it works. There is no shortcut. A compiler that produces correct, fast code for a non-trivial language really does go through something close to these seven phases, sometimes naming them differently, sometimes interleaving them, but conceptually the same. Knowing what each phase does, what it consumes, what it produces, and *why* it exists — that is the goal of the rest of this book.

Key insight

Seven phases, each producing an artifact that is closer to executable form than the last and further from human form. The compiler is not a mysterious transformation from one shape of text to another; it is a disciplined sequence of small, understandable, individually-checkable transformations. Every chapter of this book opens one of those phases and shows what it really does.

1.9 What the rest of the book does

We have walked our anchor program from text to a number, naming each phase as it went past. The rest of the book is essentially fifteen chapters' worth of stopping at each phase and looking around. Before we start that, it is worth saying out loud how we plan to spend the next five hundred pages, in narrative form rather than as a table of contents.

The very next chapter zooms out and shows the same pipeline from a higher altitude: how the phases fit together as a software system, what the FRISC machine actually looks like, why FRISCCc is written in Java, and which design decisions cascade through the rest of the codebase. Once that orientation is done, Part II opens up the front end one phase at a time: a chapter on tokenisation that explains finite-state machines from the inside, a chapter on parsing that develops the LR(1) algorithm from first principles, a chapter on semantic analysis that builds a type system for our C subset and proves a soundness result, and a chapter on the intermediate representation that justifies its design decisions one by one.

Part III is the back end: optimisation, code generation, runtime support, and the simulator. Optimisation gets a long chapter because it has to — sixteen passes, each with a correctness argument and an intuition for when it actually helps. Code generation gets a long chapter for a different reason: lowering IR to FRISC asm forces every abstraction the front end relied on to become a concrete pattern of instructions, and seeing each abstraction realised is one of the most clarifying experiences in compiler construction. Part IV takes a fun detour: we show how to *interpret* the IR directly without lowering it at all (which gives us a slower but vastly simpler implementation strategy), and then we design a small bytecode

virtual machine to make that interpreter run faster. Part V closes the book with case studies on real programs from the FRISCC test suite, a chapter on measured performance, and a final chapter on where one might take all this — adding floats, structs, a proper IR like SSA, a real target machine, a new front end for a different language.

If the journey works, by the time we reach the back cover, the black box from the opening paragraph of this chapter will be open, and you will have read enough of a real compiler to be able to modify it, port it, or write your own. Let us start, in [Chapter 2](#), by looking at the system as a whole.

Practice

Exercise 1.1 (Conceptual · ★☆☆)

Replace the body of the anchor program with `return 5 + 3 * 2;` and predict, before running anything, what the optimised IR ([Listing 1.7](#)) is going to look like. How many instructions do you expect? What numeric value will the single remaining temporary hold? Now run `./run.sh --ir --01 program.c` and check your prediction against the output.

Exercise 1.2 (Conceptual · ★☆☆)

[Listing 1.6](#) contains two instructions; [Listing 1.7](#) also contains two instructions. Read the two listings side by side. Which specific instruction has changed, in what way, and what is the most plausible name for the optimiser pass that made the change? You do not need to read any compiler source code to answer; the names of the sixteen passes are listed in [Chapter 7](#) when we get there, and one of them describes exactly the transformation visible here.

Exercise 1.3 (Applied · ★★★)

Find the `HALT` instruction in [Listing 1.8](#). Where does it appear, and why is it where it is rather than at the end of the file? The simulator processes `HALT` differently from any other instruction; describe in two sentences what the simulator's fetch-decode-execute loop does when it encounters `HALT`, and what would happen if `HALT` were never reached.

Notes and further reading

The seven-phase decomposition presented in this chapter is the *canonical* one for what the literature calls a “classical” compiler. The phase names vary slightly between textbooks — some authors distinguish a separate *intermediate code generation* step from *optimisation*, others fold semantic analysis into parsing — but the underlying idea, that a compiler is

a pipeline of small typed transformations, has been standard since the late 1970s. The reference that still defines the vocabulary for most working compiler writers is Aho, Lam, Sethi, and Ullman’s *Compilers: Principles, Techniques, and Tools* [2], universally known as the Dragon Book. Its long-form treatments of lexing, LR parsing, and dataflow are the canonical accounts, and we will cite it repeatedly.

For readers who want a more engineering-oriented companion — one that talks about real compiler back ends and the implementation choices a production team faces — Cooper and Torczon’s *Engineering a Compiler* [18] is the standard reference, with a particularly good treatment of register allocation and modern optimisation pipelines. Muchnick’s *Advanced Compiler Design and Implementation* [62] is older but still unrivalled as an encyclopaedia of optimisation techniques.

The book you are reading owes its style to a third tradition. Robert Nystrom’s *Crafting Interpreters* [67] showed — compellingly — that you do not need to write about compilers the way the Dragon Book does, and that a friendly, first-person, build-along narrative loses none of the rigour and gains an enormous amount of momentum. *Crafting Interpreters* stops short of native code generation (it ends at a bytecode VM); this book picks up roughly where Nystrom leaves off, and adds the back-end machinery, the intermediate representation, and the optimisation passes that get us all the way to FRISC machine code. If you have read Nystrom and enjoyed it, you should feel at home here. If you have not, you might enjoy reading the two side by side: he stops where we begin, and the overlap is small enough that the two books complement rather than duplicate each other.

For the deeper formal foundations — semantics, type systems, verification — we will dip into Pierce’s *Types and Programming Languages* [69] and Winskel’s *The Formal Semantics of Programming Languages* [84] in Chapters 5 and 6. Readers comfortable with mathematical reasoning will get more out of those chapters by reading the relevant sections of Pierce and Winskel first; readers who would rather build the compiler and absorb the formalism through the code can safely skip ahead and treat the proofs as commentary.

One last note. The FRISC architecture itself is not invented for this book. It is the educational ISA developed at the Faculty of Electrical Engineering and Computing at the University of Zagreb, where it was for years the primary teaching machine of the *Computer Architecture 1* course [9] before later being superseded there by the ARM architecture; the C subset we compile is the one defined by the *Prevodenje programskih jezika* (PPJ; in English, Programming Language Translation, PLT) course [27]. FRISCcc started life as the author’s project for that course and grew from there. The full FRISC specification is reproduced in Appendix B, and the PLT-FER C subset is documented exhaustively in Appendix D.

Chapter 2

The shape of a compiler

“Simple things should be simple, complex things should be possible.”
— Alan Kay

In the last chapter we put a seven-line program in one end of FRISCcc and took the integer fifteen out of the other. We named each phase as the program flowed past — lexer, parser, semantic analyser, IR builder, optimiser, code generator, simulator — and we lingered just long enough on each one to see its output before moving on. That was the tour; this chapter is the map. We are going to stand back from the program and look at the compiler architecture as a piece of software: what each phase actually owes the next, why an intermediate representation earns its place in the middle of the pipeline, what the FRISC machine looks like as a thing the back end has to produce code for, and why the whole compiler is written in Java instead of one of the half-dozen other languages that might have done.

None of this is a fresh introduction. We already have a vocabulary, and we have two pictures from Chapter 1 (Figure 1.1 and Figure 1.2): one tells us which phase runs next, and the other tells us what the program *is* between each phase. Both are about to start earning their keep. The pipeline view answers “what runs next”; the artifact view answers “what does the next phase get to assume”. Almost every design decision in the rest of this book turns on the second question, and we are going to spend this chapter unpacking it.

By the end of these twenty-five pages we will have one more picture and one more idea. The picture is the FRISC itself — eight registers, a flat memory, an ALU, a fetch-decode-execute loop — and the idea is the *artifact contract*: the typed promise each phase makes to the phase downstream of it. The contract is the reason the seven boxes of Figure 1.1 can be developed, tested, and reasoned about in isolation. It is also, as we will see in Part II, the reason Chapter 3 can talk about tokenisation without saying a word about parsing, and Chapter 4 can talk about parsing without saying a word about types.

2.1 From a string of characters to a stream of cycles

Let us start by reading Figure 1.1 differently from how we read it in Chapter 1. There we paid attention to the rounded navy boxes — the phases themselves — and we walked through them one at a time. This time, pay attention to the *arrows* between the boxes. Each

arrow is labelled with an artifact: “tokens”, “parse tree”, “typed tree”, “IR”, “optimised IR”, “FRISC assembly”. Those labels are not decoration. They are types. The arrow says that whatever the upstream phase produces, the downstream phase is entitled to assume about that artifact.

The contract is sharper than “the lexer produces tokens”. The lexer produces, to be precise, a finite sequence of tokens that ends in a distinguished end-of-file marker, contains no token whose lexeme failed to match any rule, and carries a source location on every entry. The parser, downstream, is entitled to assume all of that. The parser does not have to defend itself against an unterminated string literal or a non-ASCII control character in column thirty-three; the lexer has already rejected those programs with a diagnostic, and the parser will not see them. In return, the parser owes its own contract to semantic analysis: a tree where every non-terminal has the full set of children its grammar rule prescribes, where every terminal carries its lexeme, and where the entire program parses or it does not. There is no half-parsed tree handed downstream and no node with a missing child silently slipping past. Either the parser accepts the program and produces a well-formed tree, or it rejects the program and produces nothing.

This is what we mean by an *artifact contract*. The interface between two phases is not a function signature in Java; it is a typed promise about a data structure. The Java signature is the syntactic shadow of the contract. The interesting part lives in the prose around the signature, and in the test suite that exercises the corners.

A compiler is, formally, a partial function $\mathcal{C} : \mathcal{S} \rightharpoonup \mathcal{T}$ from a source language $\mathcal{S}$ to a target ISA $\mathcal{T}$. It is *partial* because not every input string is a valid source program; the compiler must be allowed to refuse. When it accepts, it produces a target program; when it refuses, it produces a non-empty list of diagnostics. The property we want $\mathcal{C}$ to have is *semantic preservation*: for every accepted source program p and every input ι , $\llbracket p \rrbracket_{\mathcal{S}}(\iota) = \llbracket \mathcal{C}(p) \rrbracket_{\mathcal{T}}(\iota)$. The compiled program does what the source program said it would do. The seven phases of FRISCcc are seven smaller partial functions whose composition is $\mathcal{C}$, and proving that the composition is semantics-preserving reduces to proving that each phase is. This is the framing the rest of the book uses, and the proofs of semantic preservation pass-by-pass live in [Appendix G](#).

The contracts have a useful property that has nothing to do with their content: they let us *compose* the pipeline without integration testing. If the lexer is correct in isolation, and the parser is correct in isolation, and the parser’s preconditions are exactly the lexer’s postconditions, then their composition is correct without any further argument. We can develop and test each phase against fixtures that satisfy the upstream contract, instead of running the whole pipeline to find out whether the new phase works. In a compiler with seven phases this is not a luxury; it is the reason any of this is tractable. Eight registers, four condition flags, and sixteen optimisation passes are simple in isolation; their combinatorial product is decidedly not.

Key insight

Each phase consumes a typed artifact and produces a typed artifact. The artifact's type is more than its Java class; it includes the invariants the upstream phase promises to enforce. Phases stay testable in isolation precisely because their contracts compose: the parser does not have to defend itself against malformed tokens because the lexer's contract forbids them.

There is something almost architectural about this picture. The contracts are the interfaces; the phases are the implementations. We have, in effect, written down a small operating system for the problem of transforming a C program into machine code, and the phases are the processes that run on it. The contracts pin down what the processes are allowed to assume about each other; the rest is local detail. When the pipeline survives a refactor — say, we replace the LR(1) parser with a hand-written recursive descent — it survives because the contract did not change, even though the implementation did.

What does this mean concretely for the artifacts of Chapter 1? Take the tokens of [Listing 1.2](#). The lexer's contract with the parser includes, at minimum: every token has an integer kind, a string lexeme, and a source location; the stream is finite and terminates in a distinguished end-of-input token; no two adjacent tokens are separated by a lexeme the lexer was supposed to consume but did not. None of these are written down in the type `List<Token>`; they live in the test suite, in the lexer's specification in `config/lexer_definition.txt`, and in the eight pages of Chapter 3 that justify why the maximal-munch construction satisfies them. The type is the carrier; the contract is what makes the carrier trustworthy.

Now look at the parse tree of [Listing 1.3](#). The parser's contract with semantic analysis is that the tree is a faithful derivation: every internal node corresponds to one application of one grammar rule, every leaf corresponds to one token from the lexer's output, and the entire tree spans the entire token stream. There are no orphaned subtrees, no children inserted out of order, no non-terminals with a partial child set. Semantic analysis can walk the tree pre-order, in-order, or post-order with the confidence that whatever it sees was put there by an application of a grammar rule that the parser's specification permits. The grammar in `config/parser_definition.txt` is, in this sense, the *type* of the parse tree — richer than any Java declaration, but just as binding.

The pattern repeats at every phase boundary. Semantic analysis produces an annotated tree; its contract with IR generation is that every expression node carries a type and an l-value flag, every name use carries a pointer to a declaration, and every return statement has been verified to match the surrounding function's signature. IR generation produces a sequence of three-address instructions; its contract with the optimiser is that every temporary is typed, every basic block ends in a control-flow instruction, and the program has a unique entry point. The optimiser produces another IR program with the same contract; the only difference is that the IR is, hopefully, smaller or faster. Code generation produces FRISC assembly; its contract with the simulator is the FRISC ISA and the FRISCcc calling convention, both of which we will spell out in the next two sections.

If we squint at all six handoffs together, the compiler driver is almost embarrassingly simple.

It is a six-step pipeline whose only job is to thread the artifact from one phase to the next, check for diagnostics in between, and bail out on the first error. [Algorithm 2.1](#) writes it out.

Algorithm 2.1: The FRISCcc pipeline driver: read source, lex, parse, type-check, lower, optionally optimise, and emit FRISC assembly, bailing out the moment any phase produces an error diagnostic.

Input: source-file path p , options opt (optimisation level, dump flags)

Output: FRISC assembly file, or a non-empty list of diagnostics

```

1  $src \leftarrow$  read file  $p$  ;
2  $tokens \leftarrow$  lex( $src$ ) ;
3 if errors( $tokens$ )  $\neq \emptyset$  then return diagnostics;
4  $tree \leftarrow$  parse( $tokens$ ) ;
5 if errors( $tree$ )  $\neq \emptyset$  then return diagnostics;
6  $typed \leftarrow$  semantic( $tree$ ) ;
7 if errors( $typed$ )  $\neq \emptyset$  then return diagnostics;
8  $ir \leftarrow$  lowerToIR( $typed$ ) ;
9 if  $opt.level \geq 1$  then  $ir \leftarrow$  optimise( $ir, opt$ );
10  $asm \leftarrow$  codegen( $ir$ ) ;
11 write  $asm$  to compiler-bin/a.out ;
12 return success ;
```

The body of [Algorithm 2.1](#) is, near-literally, the body of the `PipelineRunner.run` method in `cli/`. Every artifact named on the right-hand side of an arrow corresponds to a Java type the previous phase produces and the next phase consumes; every error check corresponds to a single `Diagnostics.hasErrors()` call. The driver itself is fifty lines of Java; the work is all in the phases.

What happens when a phase *cannot* satisfy its contract? It refuses, and it emits a diagnostic. Diagnostics in FRISCcc are a first-class artifact: they carry a source location, a stage tag (which phase produced them), and a severity. Once a phase emits any diagnostic with severity `ERROR`, downstream phases are skipped and the compiler exits with a non-zero status. This is the *fail-fast* discipline that lets the contracts above mean what they say. A phase never receives a malformed artifact at run time; if it does, that is a compiler bug, not a user error, and Chapter 14 discusses what we do about it.

Composability bought us testability. It also bought us replaceability. Suppose we decide, two years from now, that we want FRISCcc to accept not C but a small Pascal. We do not have to rewrite the optimiser, the code generator, or the simulator. We rewrite the lexer (Pascal has different tokens) and the parser (Pascal has a different grammar) and the semantic analyser (Pascal has different scoping rules), and we arrange for the IR generator to translate Pascal semantic trees into the same IR the rest of the pipeline already understands. Five phases out of seven survive untouched. The cost of the contracts at

design time is amortised many times over at maintenance time.

The same logic runs the other way. Suppose we want to retarget FRISCcc from FRISC to, say, RISC-V. The front-end phases survive; we replace the code generator and the simulator and possibly tweak a couple of optimisation passes that knew too much about FRISC's two-instruction multiplication helper. Two phases out of seven change. Neither of these stories is hypothetical; they describe, almost exactly, what GCC and LLVM do for a living. The reason they can is that they invested, early, in the artifact in the middle of the pipeline. We are going to spend the next section explaining why that investment pays off.

2.2 Why the IR sits in the middle

Suppose we had decided, when designing FRISCcc, to skip the intermediate representation entirely. The semantic analyser would have produced a typed tree, and the code generator would have walked that tree directly to emit FRISC assembly. There is nothing absurd about this; many real compilers, especially small ones, have done exactly this. PCC, the Portable C Compiler from Bell Labs in the late 1970s, was structured roughly this way. Wirth's compilers for Modula-2 and later Oberon were also structured roughly this way. The earliest versions of GCC came close.

The trouble appears when we ask what happens if we want to add a second target. Now we need a code generator that walks the typed tree to emit RISC-V instead of FRISC. The two code generators share roughly zero code: their input is the same tree shape, but the trees carry type information that is convenient for the front end and inconvenient for the back end. The FRISC code generator knows about FRISC addressing modes, the FRISC calling convention, the FRISC ABI for multi-word arithmetic; none of that is in the typed tree. The RISC-V code generator knows about a different set of all of those things. Each generator is a monolithic block of code that simultaneously understands the source language and the target machine. Adding a third target means writing a third monolith.

And it is worse than monoliths. Suppose we also want to add a second *source language* — say, the Pascal hypothesis from the previous section. Now we have two typed-tree shapes (one for C, one for Pascal) and two target machines (FRISC and RISC-V), and we need four code generators: C-to-FRISC, C-to-RISC-V, Pascal-to-FRISC, Pascal-to-RISC-V. The pattern generalises: M source languages and N targets require $M \times N$ code generators, each of which has to re-implement all the optimisations the others already know about. This is the engineering disaster that compiler writers have been trying to escape since the 1960s.

The escape is an intermediate representation. Sit a single, small, regular language in the middle of the pipeline. The front ends (*plural*) translate their respective source languages into IR; the back ends (*plural*) translate IR into their respective target machines. Now M source languages need M front ends, N targets need N back ends, and the total cost is $M + N$ instead of $M \times N$. The optimisations live on the IR, not on the source-to-target path, which means every front end and every back end inherits them for free. With six

source languages and six targets, the arithmetic moves from thirty-six monoliths to twelve smaller programs and one shared middle. The savings compound.

The architectural pattern of a shared IR is at least as old as IBM's Project Stretch in the late 1950s. It was articulated in modern form by Frances Allen, John Cocke, and the IBM PL.8 team in the 1970s, and elevated to its current orthodoxy by GCC in the 1990s and LLVM in the 2000s. Lattner and Adve's foundational LLVM paper [53] describes the modern restatement: a strongly-typed virtual instruction set that is rich enough to compile from many languages and concrete enough to lower to many machines. Cooper and Torczon [18] and Muchnick [62] both devote substantial early chapters to the design space of IRs; both are worth reading before designing your own.

There is a second alternative we should rule out for completeness: folding semantic analysis into parsing. This was popular in early Pascal compilers, where a single recursive-descent pass simultaneously parsed the program, built the symbol table, and type-checked expressions. The advantage is one fewer phase and a smaller working set in memory. The disadvantage is that the grammar and the type system are governed by different kinds of rules, and mixing them in a single recursive-descent pass tends to produce code that is hard to read and harder to change. When we eventually want to add, say, implicit conversions or operator overloading, every rule in the type system has to be threaded through the parser by hand, and the parser stops being something we can describe with a context-free grammar. FRISCCc keeps the two phases separate not because it is mandatory but because it is hygienic; the parser is a derivative of the grammar, and the semantic analyser is a derivative of the type system, and the two derivatives are easier to maintain when they do not share a stack frame.

That leaves the design question of what the IR should look like, which is, in a real sense, the central design question of compiler engineering. FRISCCc's IR is a typed three-address code: a linear sequence of instructions, each with at most three operands, organised into basic blocks. We saw a small specimen in [Listing 1.6](#). Chapter 6 will describe the full grammar; for now the relevant fact is just that the IR sits at exactly the right level of abstraction to be useful from both ends. From the front end's side, it is concrete enough that lowering a C expression to it is a near-mechanical walk over the typed tree. From the back end's side, it is abstract enough that one IR instruction usually maps to a small handful of FRISC instructions, with the messy details (how to encode constants, how to address local variables, how to call helper routines for multiplication) hidden inside the lowering rules.

The IR's three-address shape is a deliberate choice. The other common shape is *stack-based* code, in the JVM bytecode tradition: every instruction pops its operands off an evaluation stack and pushes its result back. Stack code is more compact (no need to name operands) and easier to interpret directly (the interpreter is a simple stack machine), but it is harder to optimise: most optimisations want to talk about which value is produced by which instruction, and stack code keeps that information implicit. Three-address code makes the dataflow explicit at the cost of giving every instruction a name. The optimisation passes of

Chapter 7 will spend most of their time chasing those names. The bytecode VM of Chapter 12 will revisit the stack-based representation as an exercise in compactness.

A third common IR shape is *static single-assignment form*, or SSA, in which every temporary is defined exactly once and any control-flow join that needs to merge two definitions is bridged with a special *phi-function*. SSA is the canonical IR of modern production compilers — LLVM, GCC’s middle end, the V8 JavaScript engine all use it — because it makes dataflow analysis nearly trivial: the definition of a temporary is, by construction, the unique instruction with that name on its left-hand side. FRISCcc deliberately does not use SSA. The reason is pedagogical: the analyses our sixteen passes do are small enough to fit in the reader’s head without phi-functions, and the construction of SSA form, while well-understood, is intricate enough that introducing it would consume a chapter we would rather spend on actual transformations. The classical reference for those who want it is Cytron, Ferrante, Rosen, Wegman, and Zadeck’s 1991 paper [21].

Why dwell on this for so long? Because the IR is the centre of gravity of the compiler. The seven phases of Figure 1.1 look symmetric in the diagram, but they are not symmetric in importance. The front end exists to produce the IR; the back end exists to consume it. Every decision the IR designer makes — what types the IR carries, what control-flow primitives it offers, how it names its temporaries — propagates outward into the front end’s lowering rules and the back end’s selection patterns. Get the IR right and everything around it is straightforward. Get it wrong and the back end becomes a forest of special cases.

We have promised ourselves that we will get it right, or close enough that the rest of the book can demonstrate the consequences. To do that we need to know what the back end has to produce, which is to say, we need to know the FRISC.

2.3 The FRISC, in one breath

Here is the machine we are compiling to. Eight registers. A single ALU. A fetch-decode-execute loop against a flat 1 MiB of memory. Figure 2.1 shows the whole processor at a glance; the rest of this section fills in what each box does and why it matters to the code generator.

The FRISC — short for *FER Reduced Instruction-Set Computer* — is a 32-bit educational architecture developed at the Faculty of Electrical Engineering and Computing at the University of Zagreb in the early 2000s. For over a decade it was the teaching machine of the *Computer Architecture 1* (*Arhitektura računala 1*) course, until ARM replaced it there in 2015. Its lineage runs through the MIPS family that Hennessy and Patterson popularised in their textbook a decade earlier [35, 68]. The descent is visible in every line of

the spec: load/store architecture, fixed instruction width, small register file, no microcode, no operating system, no memory management unit, no floating point unit, no surprises.

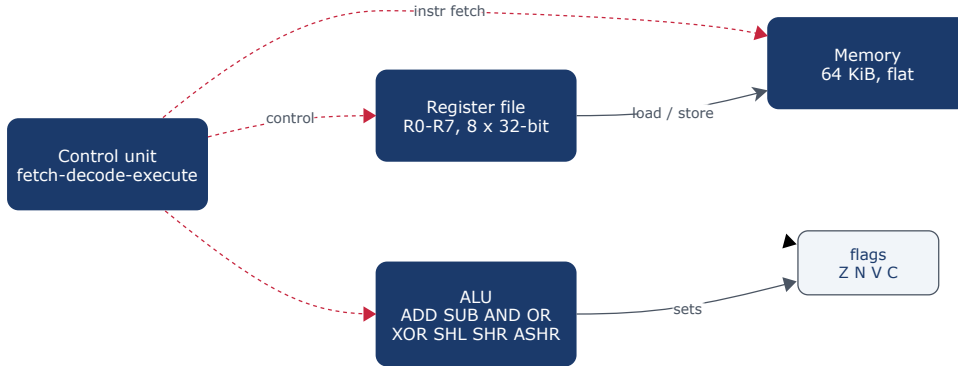


Figure 2.1: Eight 32-bit registers (R0–R7) feed an ALU; a control unit drives fetch-decode-execute against a flat 1 MiB memory; condition flags (Z, N, V, C) gate the conditional branches. No cache, no pipeline, no operating system.

The register file is the starting point: R0 through R7, each holding a 32-bit word. Most are general-purpose, but two have conventional roles by convention rather than by hardware — R5 is the frame pointer, R7 is the stack pointer — and R6 is, again by convention, where function results are returned. The hardware does not enforce any of this; the assembler does not enforce any of it; the calling convention is a social contract, and the code generator is its only enforcement. We will come back to that contract in a moment.

The code generator lives or dies on three concerns: moving values, computing values, and changing the flow of control. The FRISC’s instruction set is small enough that we can map each concern onto one or two families and be done.

Everything that *moves* a value goes through MOVE (register to register, or constant to register), LOAD (memory to register), or STORE (register to memory). The load and store instructions come in 32-bit, 16-bit, and 8-bit variants for the char-sized and short-sized data the C subset supports. Everything that *computes* goes through the arithmetic group: ADD and SUB for integer addition and subtraction, AND, OR, XOR for bitwise operations, and SHL, SHR, ASHR for shifts. Conspicuously absent are MUL, DIV, and MOD; the FRISC has no hardware multiplier or divider, and Chapter 9 will spend an entire section on the helper routines that fill the gap. Everything that *branches* goes through JP (unconditional jump) or one of its conditional cousins (JP_Z, JP_NZ, JP_C, and so on, one per flag combination); function calls go through CALL and RET; and HALT is how every program eventually terminates.

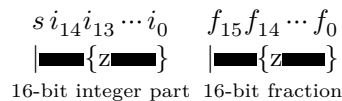
Memory comes next. The FRISC has a flat 1 MiB address space, addressable as bytes but most naturally manipulated as 32-bit words. There is no virtual memory, no paging, no protection. The whole address space is read-write; the only thing that distinguishes “code memory” from “data memory” is which addresses we choose to put which artifacts at. The stack starts high in memory — at address 40000, as we will see when we sketch the calling

convention — and grows downward; the heap, if we had one, would start at the bottom and grow upward, but FRISCcc does not implement a heap because the C subset it accepts has no dynamic allocation. The first few words of memory are conventionally reserved for the interrupt vector, which the simulator initialises but which FRISCcc programs do not use.

The arithmetic, logic, shift, and compare instructions all write the four condition flags — Z (zero), N (negative), V (overflow), C (carry) — and the conditional jumps read them. There is no per-instruction predication; the four condition flags are global state held in the status register, so any ALU instruction between a comparison and its branch silently clobbers the branch condition. This is the same problem that hardware pipeline designers call a data hazard, except we are the pipeline; the code generator carries the responsibility for keeping the write-then-read window clean. Chapter 9 will spend a couple of pages on the resulting code patterns.

That is essentially everything the hardware does. The architecture is designed to be teachable in one semester to undergraduates who have never seen a processor before, and the simplicity shows: a complete description of the FRISC fits in twenty pages, every one of which we have reproduced in Appendix B. The down side is that some things the language wants are not natively supported and have to be synthesised out of what the hardware does offer. Multiplication is the canonical example: $a * b$ in C has to become a call to a helper routine that multiplies by repeated addition (with shifts to make it fast). Division and modulo are similar. So is floating-point arithmetic, and the way FRISCcc handles *that* deserves its own sidebar.

The FRISC has no floating-point unit. FRISCcc nonetheless accepts `float` as a type. The trick is a 32-bit *Q16.16 fixed-point* representation: the top 16 bits store the integer part of the number, the bottom 16 bits store the fractional part as a binary fraction (1/2, 1/4, 1/8, and so on). The bit layout is



where s is a sign bit (the value is interpreted as two's complement). Addition and subtraction work bit-for-bit with the integer instructions; multiplication and division need to scale by 2^{16} , which is done by the helpers `F_Fmul` and `F_Fdiv`. Conversions between `int` and `float` go through `F_I2F` (shift left 16) and `F_F2I` (arithmetic shift right 16). The representable range is roughly ± 32767.999 ; the resolution is $2^{-16} \approx 1.5 \times 10^{-5}$. This is a classical engineering trade-off: real hardware floating-point is expensive in transistors, but most embedded computations can afford fixed-point precision. The FRISC's deliberate omission of an FPU turns “how to compile floats” from a hardware question into a software-engineering one, which makes it instructive.

We promised a calling convention sketch, and here it is. The FRISC ABI as FRISCcc implements it is sparse enough to fit in a paragraph. The stack pointer `R7` is initialised to

address 40000 (decimal) at program start and grows downward; the frame pointer R5 marks the base of the currently-executing function's stack frame; the return-value register R6 holds the value the callee placed there before returning. Argument registers R1 through R4 hold the first four arguments of a call; arguments beyond the fourth (if any) are pushed onto the stack by the caller. The expression-result register R0 holds the value of the most recent expression evaluation during code generation. It is caller-saved: a callee may clobber it freely, so the caller must spill it before issuing a CALL if it still needs the value afterward. Function prologues push the old R5, set the new R5, and decrement R7 to allocate local-variable space; function epilogues reverse the sequence. Every function in every program ever compiled by FRISCCc obeys this contract, and the test of whether a compiler is correct includes the test of whether its calling convention is consistent with itself.

The helper routines are a brief story. Because the FRISC lacks hardware multiplication, division, modulo, and floating-point support, the runtime ships a handful of helper labels: F_MUL, F_DIV, F_MOD, F_FML, F_FDIV, F_I2F, and F_F2I (the ones we have already met), along with L_BOUNDS_ERROR for runtime bounds violations. Each helper is a short FRISC assembly routine; each obeys the FRISCCc calling convention; each is linked into every compiled program by the code generator. Their assembly source is in Appendix B and the rationale for each is in Chapter 9 when we discuss runtime support.

The FRISC, all told, is a small machine. It has no surprises, no microarchitectural quirks, no out-of-order execution that might reorder our carefully-scheduled instructions. The price of that simplicity is that some things — multiplication, floats, anything the C standard takes for granted — have to be implemented in software. The benefit is that everything we compile to it, we can trace through, instruction by instruction, on a piece of paper. That trace-ability is what makes this book possible. Without it we would be writing a paper on register allocation; with it we can write a book on what a compiler actually does, line by line.

The hardware is mapped. We have the input language, the output ISA, and the abstract pipeline that connects them. What remains is the question of which programming language we will use to actually build the thing.

2.4 Why we are writing it in Java

Here is the shape of every IR-walking pass in this book:

```

1  sealed interface Instr permits Add, Move, Jump { }
2  record Add(Temp dst, Temp lhs, Temp rhs) implements Instr { }
3  record Move(Temp dst, Temp src) implements Instr { }
4  record Jump(Label label) implements Instr { }
5
6  String pretty(Instr i) {
7      return switch (i) {
8          case Add(var d, var l, var r) -> d + " = " + l + " + " + r;

```

```
9      case Move(var d, var s)      -> d + " = " + s;  
10     case Jump(var label)        -> "jp " + label;  
11   };  
12 }
```

The `switch` is exhaustive: the Java compiler refuses to let us leave out a case, and if we ever add a fourth instruction record, every existing pass refuses to compile until we have decided what it should do with the new node. This is Java 21's sealed-interface pattern matching, and it is the single feature that makes Java a comfortable language for writing a compiler in. Without it, FRISCcc would have looked like a 2008 Java codebase: visitor patterns, a double-dispatch method per node type, and a dispatch ceremony nobody enjoyed. With it, every pass is a single `switch` over a closed algebraic data type, and the compiler tells us when we have forgotten one of the cases.

The choice of implementation language is not religious. We picked Java for FRISCcc not because Java is the best language for compilers — it almost certainly is not — but because it is a good enough language for compilers and a much better language for the audience of this book than the obvious alternatives. The decision is small but its consequences are everywhere; this section is the story of how those consequences play out.

OCaml is the historical language of compiler writers, and for good reason: a pattern match over a sealed IR type in OCaml reads almost like the mathematical definition of the function being defined. Rust gives us the same exhaustive match with compile-time memory safety, so that an unintended aliased reference to an IR node becomes a type error rather than a Tuesday-afternoon debugging session. Haskell goes further still and lets us describe the parser as a combinator expression that mirrors the grammar line-for-line. Python with type hints would have given us the fastest path to a working prototype. All four buy expressive power that Java 1.4 simply did not have, and in another timeline this book would have been written in OCaml. We did not choose any of them, and the reason is that the source code of FRISCcc is a *teaching artifact* as much as it is an executable: code we cannot read is code we cannot learn from, and the book's audience is more likely to know Java than any of those other four.

Java has come a long way since the 1.4 days. Modern Java 21, which is what FRISCcc targets, is genuinely well-suited to writing compilers, because it has caught up with the features that compiler writers have historically left Java to find. Sealed interfaces and records combine to give us the equivalent of algebraic data types: an IR Instruction interface, sealed to permit only the dozen-or-so concrete record classes that implement it, with `switch-expression` pattern matching that the compiler verifies is exhaustive. The boilerplate of writing a tree-walking pass, which used to require a visitor pattern with a method per node type and a dispatch ceremony nobody enjoyed, is now a single `switch` expression that the compiler will not let us leave incomplete. This is the JEP 441 feature set delivered in Java 21 [11], and it makes the difference between Java being workable for a compiler and Java being pleasant for one.

The build system is Maven, the multi-module variety. Each phase of the compiler lives in its own module; the modules form a directed dependency graph that mirrors the pipeline of Figure 1.1 almost exactly.



Figure 2.2: The Maven module dependency graph of FRISCCc. Each phase of the compiler is one module: `compiler-lexer`, `compiler-parser`, `compiler-semantics`, `compiler-ir`, `compiler-opt`, `compiler-codegen-frisc`. The `cli` module at the bottom orchestrates them; `compiler-common` at the top provides the diagnostic and source-location infrastructure that every other module depends on. The graph is acyclic by construction: no module depends on a module “downstream” of it in the pipeline. This is what makes parallel `mvn` builds work, and what lets us test each phase in isolation.

Figure 2.2 earns its keep by making the modularity visible. The pipeline of arrows in

Figure 1.1 corresponds to a chain of Maven artifacts linked by `<dependency>` declarations in `pom.xml` files. The parser module depends on the lexer (it consumes the `Token` class); semantics depends on the parser; IR depends on semantics; optimisation depends on IR; the FRISC codegen depends on IR but not on anything earlier. The `cli` module sits at the bottom and depends on every phase, because its job is to orchestrate them. The shared `compiler-common` module sits at the top, depended on by every phase, because its job is to provide the diagnostic plumbing that every phase needs to report errors. The graph is acyclic, which is what allows Maven to build the modules in parallel and what lets us test each module in isolation against fixtures that satisfy the upstream contract.

The shape of Figure 2.2 is the *layered architecture* pattern under a different name. The top layer (`compiler-common`) is the kernel of types used by everyone. The middle layers are the phases. The bottom layer (`cli`) is the orchestrator. Every dependency points upward; there are no cycles. The pattern is the same one that underlies Spring, every JDK module, and most large Maven projects. The reason the pattern keeps showing up is that it is the cheapest way to preserve isolation in a system whose pieces want to talk to each other. Whether you call it “hexagonal architecture”, “ports and adapters”, “onion architecture”, or just “layered architecture”, the underlying invariant — “dependencies flow one way” — is what makes any of those patterns work.

The honest case against Java is verbosity. A record class with three fields requires more keystrokes in Java than in OCaml, Rust, or Haskell. A sealed interface with a dozen implementations is a longer file in Java than it would be in a language with first-class sum types. We have made our peace with this. The book uses small excerpts from the actual source code, never whole class files; the verbosity is hidden in the parts we skip over. Anyone who opens the compiler’s source repository will find more boilerplate than the book shows, and we are not going to pretend otherwise.

In exchange we get a JVM with a sane garbage collector, which is a quietly substantial gift. AST nodes, IR instructions, and CFG basic blocks are objects that get allocated by the millions during a compile, and tracking their lifetimes by hand would consume more of our attention than the lifetimes deserve. We do not write `free()` anywhere in FRISCcc; the GC keeps the memory profile sane and the working set small enough that the entire compiler fits in a few megabytes of resident memory. For a teaching codebase this is the right trade.

We also get tooling. `mvn test` runs the entire test suite in under two minutes; `./build.sh` produces a single executable JAR; the IDE story (IntelliJ, Eclipse, VSCode) is mature enough that any reader can open the project and start stepping through the code with a debugger. None of this is glamorous; all of it is the kind of infrastructure that a compiler project lives or dies by.

There is one externality. The FRISC simulator we use to actually run compiled programs — `friscjs` [88] — is not Java; it is JavaScript, written by the FER teaching lab in the early 2010s and ported repeatedly since. We vendor it in `node_modules/` because it is the canonical reference simulator for the FRISC and we do not want to drift from the spec

by writing our own. The language boundary is a deliberate experimental control: `friscjs` is the oracle, `FRISCcc` is the system under test, and the two communicating over an assembly-file interface makes their independence visible. In Chapter 11 we will write our own IR interpreter in Java and in Chapter 12 a bytecode VM; both will be cross-checked against `friscjs` on the same input programs, and the language boundary is what gives us confidence in the cross-check.

A historical note. The architecture in which a compiler is built as a collection of small, single-purpose modules with sharp interfaces is older than Java. The pioneering example is Niklaus Wirth’s Modula-2 and Oberon compilers, which both shipped as self-hosted binaries built from explicitly modular source [85]. Wirth’s notes on those compilers — still in print, still freely available — are an excellent education in what an implementation language buys and costs a compiler writer. The reader who wants a non-Java perspective on `FRISCcc`’s design choices should read Wirth’s monograph back-to-back with this book; the contrast is illuminating.

The shape is mapped, the materials are chosen, the tooling is in place. We have spent a chapter standing back from the compiler and looking at it as a piece of software: seven phases connected by typed artifact contracts, an intermediate representation in the middle that decouples the front from the back, a small RISC machine at the end that the back end has to satisfy, and a Java-21 codebase that expresses all of this in a way we can open and step through. Now we get to start opening it. Part II begins on the next page with the first phase of the pipeline, the lexer, and the question of how a finite-state machine can recognise the difference between `int` and `intern`.

Your turn

This chapter has no companion lab of its own — the first build-along chapter is Chapter 3 — but before moving on we should make sure the companion repository is set up so the next chapter can hit the ground running. The companion is its own open-source repository [46], separate from the `FRISCcc` compiler we have been reading; clone it from <https://github.com/KarloKnezevic/frisccc-companion> into a sibling directory to this book, where the chapters refer to it as `frisccc-companion/`. Open a terminal there and run `mvn` in package mode, skipping tests, from the repository root; if the build succeeds, every module (starter and solution alike) will be compiled, and the upcoming chapters will work out of the box. Next, read the `tinyC` specification from top to bottom. It is short: five or six pages defining *tinyC*, the `int`-only subset of C that we will be writing a compiler for in the chapters ahead. Pay attention to what it omits as much as to what it includes; the omissions are what will make the project tractable for one reader over fifteen chapters. Finally, take a peek inside the `ch03-lexer/` module: notice the `starter/`, `solution/`, and `tests/` subdirectories. Every chapter module follows this shape. The starter contains the stubs you will fill in; the solution is the reference implementation you can compare against

once you are done; the tests tell you whether you got it right. When you reach the end of Chapter 3 you will return to this same module and begin the first proper build-along of the book.

Practice

Exercise 2.1 (Conceptual · ★☆☆)

We argued in [Section 2.1](#) that the lexer and the parser are usefully kept as separate phases because their contracts compose. But this is not the only possible decomposition. Argue, in three sentences, the case for merging them into a single “front-end scanner-parser” phase whose input is bytes and whose output is a tree. Then argue, in three sentences, the case against. Which side does FRISCCc take, and on which of your reasons does it rest?

Exercise 2.2 (Applied · ★★☆☆)

The FRISC stack pointer R7 is initialised to address 40000 at program start and grows *downward*. The FRISC’s addressable memory is 1 MiB, that is, addresses 0 through 1048575. Compute, in two sentences, the deepest address the stack can legally reach before it collides with the heap region or the static data region (assume the heap starts at 0 and grows upward; assume the program is small enough that static data ends at address 1000). What happens, mechanically, if a recursive function pushes a stack frame that drives R7 below the safe bound? The FRISC has no hardware stack-overflow check; what does the simulator actually see?

Exercise 2.3 (Conceptual · ★★☆☆)

Read §3.2 of Lattner and Adve’s LLVM paper [53]. The paper argues that LLVM’s IR is a three-address code in *static single-assignment* form. Two sentences: what property of three-address code does SSA preserve, and what property does SSA add? A third sentence: given the sidebar in [Section 2.2](#) on why FRISCCc does not use SSA, what does FRISCCc give up by sticking with plain three-address code, and what does it gain?

Project

Exercise 2.4 (Lab · ★★★)

Before reading Chapter 3, open the test directory of the companion repository’s `ch03-lexer` module and read the test file without looking at the starter or the solution. Each test method names a small fragment of tinyC source — `int x;`, `return 5;`, `while (a == b)`, and so on — and asserts the exact token sequence the lexer is expected to emit. Pick five test methods and, for each, write down (a) the source string, (b) the list of token kinds you would expect a correct lexer to produce, and (c) any tokens that surprise you compared with what a textbook description of C lexing would lead you to predict. Keep your predictions on paper; we will revisit them at the end of Chapter 3 once we know what the lexer actually does. The exercise is not graded by anything except your own curiosity, but the act of writing the predictions down before reading the chapter is what makes the chapter worth reading.

Notes and further reading

The seven-phase decomposition we have been using throughout this chapter, and the artifact-contract view that motivates it, is the canonical pipeline architecture for what the literature calls a “classical” compiler. The vocabulary is set by Aho, Lam, Sethi, and Ullman’s *Compilers: Principles, Techniques, and Tools*, universally known as the Dragon Book [2]. The Dragon Book treats the phases in detail and the contracts implicitly; its long chapters on lexing, parsing, and dataflow are still the standard references, though its presentation belongs to a different generation and reads, today, like a reference manual rather than a narrative. We will draw on it constantly for facts.

The engineering perspective on compiler architecture — which is much closer in spirit to the view we have taken in this chapter — is laid out in Cooper and Torczon’s *Engineering a Compiler* [18]. Their Chapter 1 is explicitly the “shape of a compiler” chapter; their Chapter 5 on intermediate representations is the best short treatment of the IR design space we know. If you read only one other book on compiler architecture alongside this one, make it that one.

For the optimisation pipeline specifically — which we will spend all of Chapter 7 on — the encyclopaedic reference is Muchnick’s *Advanced Compiler Design and Implementation* [62]. Muchnick is older than Cooper-Torczon and less narrative, but it is comprehensive in a way that few other books are; almost every optimisation FRISCCc implements has its prototype in one of Muchnick’s chapters. Read the introduction and Chapter 5 (on intermediate representations) now; we will come back to the rest.

The implementation language is Java 21, and the feature we lean on hardest is sealed-interface pattern matching as it was finalised in JEP 441 [11]. The JEP itself is short and worth reading; it explains the design rationale (exhaustiveness, type narrowing, integration with switch expressions) in less than ten pages. If you have not used pattern matching on sealed types before, write a small example before Chapter 3: a sealed interface `Shape` with `Circle`, `Square`, and `Triangle` records, and a double `area(Shape s)` method that switches

over `s`. The shape of the code is exactly the shape of every IR-walking pass in Chapters 6 through 10.

The architectural pattern of layered, module-per-phase decomposition is older than Java and predates the modern multi-module build systems. Wirth’s monograph on his own series of compilers — Modula-2, Oberon, and their descendants — is still the cleanest extant treatment of how a compiler is structured as a set of small, single-purpose programs [85]. The book is short (under two hundred pages, including the source listings of a working compiler) and the contrast with Java tooling is illuminating.

The FRISC itself is, as the chapter notes, the educational ISA developed at the Faculty of Electrical Engineering and Computing at the University of Zagreb for its *Computer Architecture 1* course [9]. Its design lineage runs through the MIPS family of academic and commercial RISC machines, which Hennessy and Patterson’s textbook [35] treats in exhaustive detail; Patterson’s 1985 *Communications* article [68] is a shorter manifesto for the RISC philosophy and is the best three-thousand-word answer to “why does a processor look the way the FRISC looks”. The full FRISC specification is reproduced in Appendix B; the PLT course materials [27] on which FRISCcc’s C subset is based are freely available from FER and are the canonical source for any ambiguity the appendix does not resolve.

Part II

The Front End

Chapter 3

Reading characters: tokenization

“The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man.”
— George Bernard Shaw, *Man and Superman*, 1903

Open a fresh C file and type `int x = 5;`. To a human reader this is a declaration: a three-letter keyword, a one-letter name, an equals sign, the digit five, a semicolon. The spaces between the pieces are decoration; we could just as well have written `int x = 5 ;` or `int x=5;` or even, perversely, `int/*hello*/x/**/=/**/5;`, and they would all mean the same thing. The mind, reading these, performs an operation so reflexive that it takes a moment to notice it is happening: it carves the stream of bytes into atoms, throws away the whitespace, and recognises each atom as a particular kind of thing.

The lexer is the part of the compiler that does that carving. Give it `int x = 5;` as a sequence of ten bytes (including the spaces), and it will give back five tokens: the keyword `int`, the identifier `x`, the equals sign, the integer literal `5`, the semicolon. Whitespace gone. Comments gone. Each surviving atom carries a kind, a lexeme, and the line and column on which it appeared — plus a slot for the symbol-table index later phases fill in — so that downstream phases can produce diagnostics if anything goes wrong. This is what we mean by tokenisation: the linear, irreversible compression of source text down to the smallest stream that still contains all the meaning.

Watch that carving happen and a whole machine comes into focus behind it. We will spend the first three sections building a small, beautiful theory of regular languages, the finite-state machines that recognise them, and the two classical constructions — Thompson’s and the subset construction — that take us from one to the other. Once the theory is in place we will add the engineering: *maximal munch* for resolving overlapping matches, lexer *modes* for handling the parts of the source that are not really tokens at all, and the actual pipeline that FRISCCc’s lexer runs from specification file to token stream. Along the way we will tokenise the chapter’s anchor program, a tiny iterative Fibonacci, and trace a few characters of its source through the machine character by character.

By the end of these thirty-odd pages we should be able to open FRISCCc’s lexer specification — the file `config/lexer_definition.txt` — and predict, line by line, what the lexer will do with any input we hand it.

3.1 Regular expressions, and why they’re not regular

The lexer’s job is pattern matching, and the best way to see what that means is to look at one pattern. The rule for a C identifier — a letter or underscore followed by any number of letters, digits, or underscores — is written

$$[A-Za-z_][A-Za-z0-9_]*$$

and FRISCCc’s lexer uses exactly that to match `i`, `result`, and `math_fibonacci_iter`. Three operators are at work in those seven symbols. Square brackets denote a *character class*: any single character drawn from the enclosed set. Concatenation, written by juxtaposition, denotes “this followed by that”. The Kleene star, the superscript asterisk, denotes “any number of repetitions, including zero”. Combined, the pattern says: an identifier is one character from the first class, followed by zero or more characters from the second. The notation is terse by design — we want something small and mechanical, because the lexer is going to compile each pattern into a recogniser and then scan megabytes of source through it. The formalism that gives us these operators, and lets us build efficient recognisers from them, is the *regular expression*.

Three operators are almost everything. The full regex algebra has one more: alternation, written with a vertical bar, denoting “either this or that”. With concatenation, alternation, and Kleene star, plus a way to talk about a single character of the alphabet, we have a notation expressive enough to describe every *regular language* — and that is the precise sense in which the regex pattern for identifiers is “regular”. Each operator builds the language of a bigger expression from the languages of smaller ones, and the inductive structure of the expression mirrors the inductive structure of the recogniser we will build for it in [Section 3.2](#).

Formally, fix an *alphabet* Σ (for FRISCCc, the printable ASCII characters plus a few whitespace bytes). A *language* over Σ is any subset $L \subseteq \Sigma^*$ of the set of finite strings over Σ . The *regular languages* are the smallest class of languages containing the empty language $\emptyset$, the singleton languages $\{a\}$ for each $a \in \Sigma$, and the empty-string language $\{\varepsilon\}$, and closed under three operations:

$$L_1 \cup L_2 \quad L_1 \cdot L_2 \quad L^*$$

namely *union* (alternation), *concatenation*, and the *Kleene closure* $L^* = \bigcup_{n \geq 0} L^n$, the set of strings that are concatenations of zero or more strings from L . A *regular expression* is a finite syntactic term over Σ built from these operators; its *denotation* is the regular language it describes. The character class $[A-Z]$ is sugar for an alternation of singletons; the question mark $r?$ is sugar for $r \cup \varepsilon$; the plus r^+ is sugar for $r \cdot r^*$. None of the sugar adds expressive power; it only makes the patterns readable.

The vocabulary is small enough to write down on one page. The expressive power, however, is sharply bounded, and the bounds are the most useful property of the formalism. There

are languages that look like they ought to be describable with a regex and are not. The set of strings of balanced parentheses — $\{ (, (()), ((())), \dots \}$ — is the classical example. No matter how clever a regex we write, we cannot match an arbitrary depth of nesting, because the recogniser has only finite memory and the language has unboundedly many distinct “depths” to remember. Programmers learn this the hard way when they try to parse HTML with regexes; computer scientists learn it the easy way by internalising the pumping lemma.

Key insight

A regular language is one that a machine with a fixed, finite amount of memory can recognise. The lexer can decide whether a string of bytes is an identifier without knowing anything about what came before it, and without keeping a counter of how many bytes it has consumed. That is exactly the property that makes the lexer run in linear time with constant memory per token, and exactly the property that puts balanced parentheses — and therefore the parsing of expressions, statements, and blocks — beyond the lexer’s reach. The boundary between the lexer and the parser is the boundary between regular and context-free.

So far we have been using “regular expression” to mean the mathematical object: a syntactic term denoting a regular language. The word means something different to most working programmers, and the difference is worth a paragraph because it explains why some regex engines are slow.

When a Perl, Python, JavaScript, or PCRE user writes a “regex”, they are writing in a notation that includes the three classical operators plus a long list of extensions: backreferences (`(w+)\1` matches a word repeated), lookahead (`foo(?:=bar)` matches `foo` only when followed by `bar`), lookbehind, named groups, non-greedy quantifiers, conditional matches, and a few features that have no name in academic literature because no academic invented them. None of these are regular in the language-theory sense. Backreferences, in particular, push the formalism well past context-free; the engine that runs them generally has to backtrack, and contrived inputs can make the running time exponential in the length of the input. The same `(a+)+$` regex that takes microseconds to compile can take hours to run against a long string of `a`s followed by a `b`.

The contrast explains an old folk theorem: `grep` is faster than `perl` when the regex is regular. The classical `grep` uses the Thompson construction we will see in [Section 3.2](#), which compiles the regex to a deterministic finite-state machine and runs in time linear in the length of the input, no matter how complex the regex. Perl’s engine is a backtracking matcher that handles backreferences and lookahead at the cost of worst-case exponential time on patterns that happen to be regular. Ken Thompson’s original 1968 paper [80] introduced the construction we will use; Russ Cox’s modern essays make the case that more engines should be using it. `FRISCCc`, like `grep`, sticks to the regular fragment. The lexer never backtracks; it never explores; it just runs.

For our purposes the upshot is liberating: by restricting the patterns to the regular fragment, we get a recogniser whose worst-case time per byte is constant, whose memory is fixed at compile time, and whose correctness we can argue from the structure of the regex. The price is that some things we might want to say about a token — “the closing quote must match the opening one”, “brackets must be balanced” — we cannot say in the regex itself. We will pay that price in the next chapter, when the parser earns its keep.

The notation, then, is fixed. What we need next is an algorithm: given a regular expression r and a string w , we need a way to decide whether w is in the language of r , and to do so quickly. The classical answer, which has hardly changed since 1968, is to compile the regex into a state machine and let the state machine do the work.

3.2 From regex to NFA: Thompson’s construction

Take the regex $a(b|c)^*$ and ask: what would a machine that accepts exactly the strings matching it look like? The pattern reads as “the character a , followed by zero or more occurrences of either b or c ”, and the machine we want turns out to be three small machines bolted together — one for the literal a , one for the alternation $b|c$, one for the Kleene-star wrapper — connected by silent “epsilon” moves that the machine takes for free without consuming any input. A machine with silent moves, with multiple transitions out of the same state on the same character, and with the convention that it follows all such transitions simultaneously, is a *nondeterministic finite automaton*: finite because it has finitely many states, automaton because it reads a character at a time and changes state in response, and nondeterministic because we treat it as exploring every reachable configuration in parallel and accepting if any path ends in an accepting state.

The construction that builds such a machine from a regex is one of the most economical algorithms in computer science. It is named after Ken Thompson, who published it in 1968 to power the regex feature of the QED text editor. The idea is inductive: for each regex operator, we give a small NFA gadget that recognises exactly the language denoted by an expression built with that operator, assuming we already have NFAs for its sub-expressions. Glue the gadgets together by the structure of the regex, and the result is an NFA for the whole thing.

To see the gadgets at work, return to $a(b|c)^*$ and build the NFA bottom-up. For each single character we draw a two-state machine with one transition labelled by that character. For the alternation $b|c$ we make a new start state with two epsilon transitions (silent moves that consume no input), one to the b machine and one to the c machine, and then merge the two accept states into a new common accept via more epsilon transitions. For the Kleene star $(b|c)^*$ we add yet another wrapping pair of epsilon transitions: one that skips the inner machine entirely (because zero repetitions are allowed) and one that loops back from the inner machine’s accept to its start (because more repetitions are allowed). For the concatenation $a(b|c)^*$ we splice the accept of the a machine into the start of the $(b|c)^*$ machine with one more epsilon transition.

Algorithm 3.1: Thompson construction of an NFA from a regex AST. Each case produces a gadget with exactly one start state and one accept state, so the sub-machines can be plugged in without inspecting their interiors. Helper `fresh()` allocates a state identifier the construction has not used before.

Input: A regex AST node r .

Output: An NFA $N(r) = (Q, \Sigma, \delta, s, f)$ with one start state s , one accept state f , and transitions δ labelled by characters of Σ or by ε .

```

1 switch  $r$  do
2   case atom  $a \in \Sigma \cup \{\varepsilon\}$  do
3      $s \leftarrow \text{fresh}()$ 
4      $f \leftarrow \text{fresh}()$ 
5     add transition  $s \xrightarrow{a} f$  to  $\delta$ 
6     return  $(Q = \{s, f\}, \delta, s, f)$ 
7   case concatenation  $r_1 r_2$  do
8      $N_1 \leftarrow \text{THOMPSON}(r_1); \ N_2 \leftarrow \text{THOMPSON}(r_2)$ 
9     add transition  $f_1 \xrightarrow{\varepsilon} s_2$  to  $\delta$ 
10    return  $N$  with start  $s_1$ , accept  $f_2$ 
11  case alternation  $r_1 \mid r_2$  do
12     $N_1 \leftarrow \text{THOMPSON}(r_1); \ N_2 \leftarrow \text{THOMPSON}(r_2)$ 
13     $s \leftarrow \text{fresh}(); \ f \leftarrow \text{fresh}()$ 
14    add  $s \xrightarrow{\varepsilon} s_1, s \xrightarrow{\varepsilon} s_2, f_1 \xrightarrow{\varepsilon} f, f_2 \xrightarrow{\varepsilon} f$ 
15    return  $N$  with start  $s$ , accept  $f$ 
16  case Kleene star  $r_1^*$  do
17     $N_1 \leftarrow \text{THOMPSON}(r_1)$ 
18     $s \leftarrow \text{fresh}(); \ f \leftarrow \text{fresh}()$ 
19    add  $s \xrightarrow{\varepsilon} s_1, s \xrightarrow{\varepsilon} f, f_1 \xrightarrow{\varepsilon} s_1, f_1 \xrightarrow{\varepsilon} f$ 
20    return  $N$  with start  $s$ , accept  $f$ 
21  case optional  $r_1?$  do
22     $N_1 \leftarrow \text{THOMPSON}(r_1)$ 
23     $s \leftarrow \text{fresh}(); \ f \leftarrow \text{fresh}()$ 
24    add  $s \xrightarrow{\varepsilon} s_1, s \xrightarrow{\varepsilon} f, f_1 \xrightarrow{\varepsilon} f$ 
25    return  $N$  with start  $s$ , accept  $f$ 

```

Algorithm 3.1 is recursive descent with the regex AST as the recursion tree: each case allocates fresh states with `fresh()`, draws the small number of transitions the case calls for, and returns the resulting machine with its single start and single accept exposed. The five gadgets the cases build are drawn in Figure 3.1; reading the algorithm next to the figure makes it clear why “one start, one accept” is an invariant worth fighting for. Pick any case and trace it on paper for a small input: the structural recursion turns the regex algebra into a circuit diagram, and the size of the resulting NFA is linear in the size of the regex.

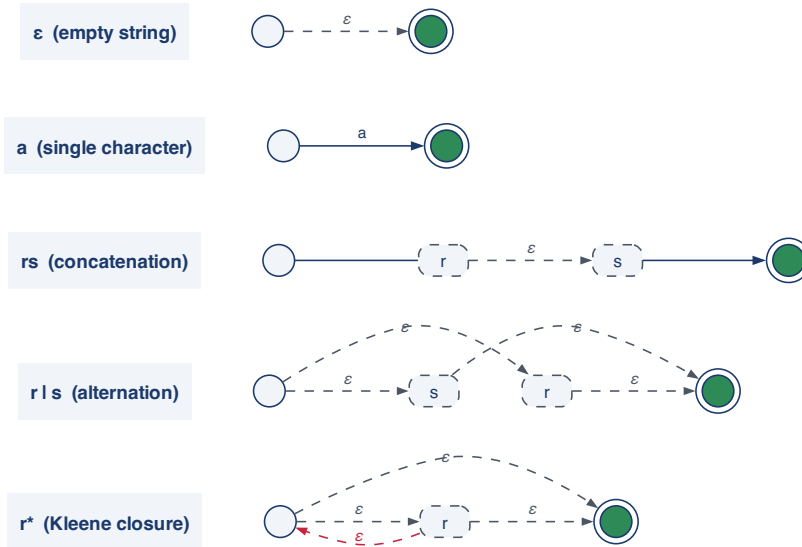


Figure 3.1: The five Thompson gadgets. Top-left: the empty string ε is a single epsilon transition between a fresh start and accept state. Top-right: a single character a is one transition labelled by a . Middle-left: concatenation rs splices the accept of the r -machine into the start of the s -machine via a silent epsilon transition. Middle-right: alternation $r|s$ wraps both sub-machines with a fresh start (with two epsilon transitions, one to each) and a fresh accept (reached from either sub-machine via epsilon). Bottom: the Kleene closure r^* wraps the r -machine with a fresh start and accept that share an epsilon transition (for zero repetitions) and a back-edge from the inner accept to the inner start (for more). Every gadget has exactly one start state and one accept state, which is what makes the inductive composition work.

The discipline that makes the gadgets compose is visible in Figure 3.1: every gadget has exactly one start state and exactly one accept state, with no incoming transitions to the start and no outgoing transitions from the accept. That invariant means we can plug any gadget into the position of any sub-expression without worrying about its internal structure.

The construction is the recursive descent of regex algebra, and each step preserves the shape that lets the next step work.

Algorithm 3.1 states the construction case by case. Its correctness theorem is the statement that, for every regex r , the language accepted by $N(r)$ is exactly the language denoted by r . The proof is by structural induction on r : each gadget accepts exactly the language denoted by its operator applied to the (assumed-correct) languages of the sub-machines. The resulting NFA has at most $2|r|$ states — one or two fresh states per operator — and at most $4|r|$ transitions, so the compilation is linear in the size of the source expression.

Two properties of the result are worth pausing on. First, the size of the NFA is linear in the size of the regex. A regex with n characters becomes a machine with $O(n)$ states and $O(n)$ transitions, which is about as tight as a compilation can be. Second, every transition is either labelled by a single character of the alphabet or is an epsilon transition. Those epsilons are how the nondeterminism arrives: an NFA state can have multiple epsilon transitions out of it, and the machine takes them all at once. The price of the simple inductive gadgets is a machine that simulates several copies of itself in parallel.

To run the NFA on an input string we have to simulate that parallelism. The standard simulation maintains, at every point in the input, the *set* of NFA states the machine is currently in — specifically, the epsilon-closure of that set, written $\varepsilon\text{-cl}(S)$, which adds in every state reachable from S by epsilon transitions alone. On each input character, we compute the set of states reachable from the current set by that character, and then take the epsilon-closure of the result. The machine accepts if and only if, at end of input, the current set contains any accepting state.

This is workable but not pleasant. Computing the epsilon-closure at every step costs us linear time per character in the worst case, which multiplied across an input of length m gives $O(nm)$ total time. For a lexer that has to process megabytes of source, that factor of n hurts. We would much prefer a recogniser that takes constant time per character, regardless of how complicated the regex was. Such a recogniser exists, and it is what we build next.

3.3 From NFA to DFA: subset construction

The NFA simulation at the end of the last section had one annoying property: at every position in the input it maintained a whole *set* of NFA states, and on every character it had to recompute that set — chase epsilon transitions, take the character transitions, chase epsilon transitions again. The work per character was linear in the size of the NFA. What if we reified those sets? Instead of recomputing the current set dynamically on every character, we could compute it once, ahead of time, for every possible character we might read from every possible set we might find ourselves in. Each reachable set would become a state of a

new machine, and each character-transition between sets would become a single outgoing edge.

That new machine is a *deterministic finite automaton*: from each state, on each character of the alphabet, there is at most one transition; there are no epsilon transitions, and there is no choice. To run a DFA, we hold a single “current state” variable, read each input character, and follow the unique outgoing transition. Each character costs us one lookup and one assignment, regardless of how many NFA states the corresponding set contains. The lexer runs in linear time per character with constant memory — the very property we wanted at the end of [Section 3.2](#), and the property we will spend this section earning.

The reification is the *subset construction*, due to Michael Rabin and Dana Scott in 1959 [73]. The work the NFA simulation was doing dynamically gets done once, at construction time; the result is a machine that runs deterministically on every input.

Let us see it on the example from the last section. We had the regex $a(b|c)^*$ and a Thompson NFA with ten states (call them $q_0, q_1, \dots, q_9$, where q_0 is the start and q_9 is the accept). The DFA construction starts with the epsilon-closure of the NFA’s start state: $D_0 = \varepsilon\text{-cl}(\{q_0\})$. From D_0 , on character a , we collect all NFA states reachable by reading a from any state in D_0 , then take the epsilon-closure of the result. Call that set D_1 . From D_1 , on b and on c , we get two more sets (which may turn out to be the same set, D_2 , if the gadgets converge). We continue this worklist procedure until no new sets appear. Each set is a DFA state; each transition we computed along the way is a DFA transition.

This is BFS on the power set of NFA states. The worklist is the frontier; new DFA states are the nodes we have not yet visited; $\varepsilon\text{-cl}$ composed with the character-step is the edge relation; and the algorithm terminates because the frontier cannot grow larger than $2^{|Q|}$. One of the lovely self-referential properties of the construction is that the objects we compute over — subsets of NFA states — are the same kind of thing as the result, namely DFA states; the search is a search through the space its own answer lives in.

The epsilon-closure step is small enough to deserve its own routine. [Algorithm 3.2](#) gives the standard depth-first implementation: push the input states on a stack, pop one at a time, and follow every outgoing epsilon transition to a state we have not yet seen.

With $\varepsilon\text{-cl}$ in hand, the worklist that drives the subset construction reduces to bookkeeping. [Algorithm 3.3](#) writes it out.

The example in [Figure 3.2](#) is small but instructive, because the two patterns `int` and `intern` share a prefix and force the construction to make a useful decision. After reading `i`, `n`, `t`, the machine is in a DFA state that is both *accepting* (the prefix `int` matches the first alternative) and *has an outgoing transition* (on `e`, toward the rest of `intern`). A lexer staring at the source `intern x`; will reach this state, notice both facts, remember that it has seen an acceptable match of `int`, and keep going to see whether it can find a longer match. That “remember and keep going” policy is exactly maximal munch, which is the subject of the next section. The DFA hands us the information; the policy decides what to do with it.

Algorithm 3.2: Epsilon-closure of an NFA state set. The procedure runs in time linear in the number of epsilon transitions reachable from S , since each state is pushed and popped at most once.

Input: An NFA $N = (Q, \Sigma, \delta, q_0, F)$ and a set $S \subseteq Q$.

Output: The epsilon-closure $\varepsilon\text{-cl}(S) = \{q \mid \exists s \in S, s \xrightarrow{\varepsilon^*} q\}$.

```

1  closure  $\leftarrow S$ 
2  stack  $\leftarrow$  a stack initialised with the elements of  $S$ 
3  while stack is not empty do
4      q  $\leftarrow$  pop(stack)
5      for each transition  $q \xrightarrow{\varepsilon} q'$  in  $\delta$  do
6          if  $q' \notin$  closure then
7              add  $q'$  to closure
8              push(stack,  $q'$ )
9  return closure

```

Algorithm 3.3: Subset construction of a DFA from an NFA, written as a breadth-first search over the power set of NFA states. Each DFA state S is a subset of Q ; each DFA transition is the epsilon-closure of the NFA's character-step applied to S . A DFA state is accepting exactly when its underlying NFA-state set contains an NFA accept.

Input: An NFA $N = (Q, \Sigma, \delta, q_0, F)$.

Output: An equivalent DFA $D = (Q', \Sigma, \delta', q'_0, F')$.

```

1   $q'_0 \leftarrow \varepsilon\text{-closure}(\{q_0\})$ 
2   $Q' \leftarrow \{q'_0\}; \delta' \leftarrow \emptyset$ 
3  worklist  $\leftarrow$  a queue containing  $q'_0$ 
4  while worklist is not empty do
5       $S \leftarrow$  dequeue(worklist)
6      for each  $a \in \Sigma$  do
7           $T \leftarrow \varepsilon\text{-closure}(\bigcup_{q \in S} \delta(q, a))$ 
8          if  $T = \emptyset$  then
9              continue
10         if  $T \notin Q'$  then
11             add  $T$  to  $Q'$ 
12             enqueue(worklist,  $T$ )
13         add transition  $S \xrightarrow{a} T$  to  $\delta'$ 
14   $F' \leftarrow \{S \in Q' \mid S \cap F \neq \emptyset\}$ 
15  return  $(Q', \Sigma, \delta', q'_0, F')$ 

```

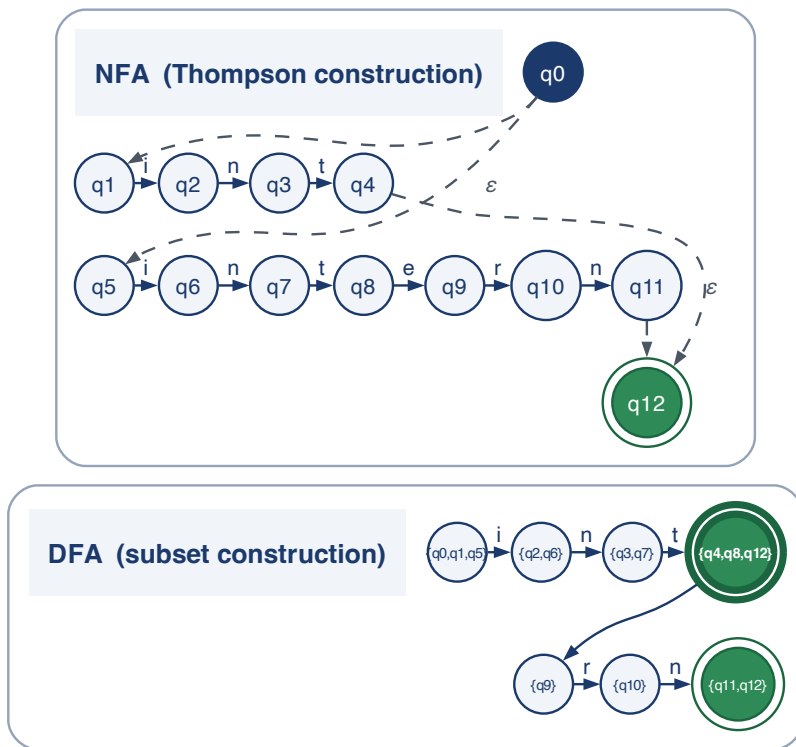


Figure 3.2: Top: the Thompson NFA for `int|intern`. Bottom: the DFA produced by subset construction. Each DFA state is labelled with the set of NFA states it represents; double circles mark accepting sets. The crucial state is the one reached after `i`, `n`, `t`: it is accepting (for `int`) and also has an outgoing transition on `e` (toward the rest of `intern`). That single state encodes everything the lexer needs to know about the conflict.

Algorithm 3.3 compiles an NFA $N = (Q, \Sigma, \delta, q_0, F)$ into a DFA $D = (Q', \Sigma, \delta', q'_0, F')$ whose states are subsets of Q , whose start state is $q'_0 = \varepsilon\text{-cl}(\{q_0\})$, whose transition function is

$$\delta'(S, a) = \varepsilon\text{-cl}\left(\bigcup_{q \in S} \delta(q, a)\right),$$

and whose accepting set is $F' = \{S \in Q' \mid S \cap F \neq \emptyset\}$. Correctness is the statement that a string w is accepted by D if and only if it is accepted by N , proved by induction on $|w|$ from the fact that $\delta'(S, a)$ tracks exactly the set of NFA states reachable from S by reading a .

The DFA can be exponentially larger than the NFA. In the worst case, an NFA with n states becomes a DFA with up to 2^n states, because every subset of Q is in principle reachable. In practice this blow-up almost never happens for lexer specifications: the regexes we write for keywords and identifiers are gentle enough that the resulting DFAs have a few hundred states at most, even when several dozen tokens share an alphabet. FRISCCc's lexer DFA for the initial mode — which has to distinguish keyword from identifier, integer from floating-point literal, single from double-character operator, and so on for fifty-odd token kinds — has fewer than five hundred states. The construction is fast and the result fits in memory without ceremony.

The DFA produced by subset construction is rarely minimal. Two states that lead to the same acceptance behaviour on every continuation can be merged, and the merging produces a smaller DFA that recognises the same language. The classical algorithm for this is John Hopcroft's $O(n \log n)$ partition refinement, published in 1971 [38]. The idea is to start with two partitions of the DFA's states (accepting and non-accepting) and repeatedly split any partition whose states do not all transition into the same partition on some character. When no more splits are possible, each remaining partition becomes a state of the minimal DFA. FRISCCc deliberately skips this step: its lexer pipeline is Thompson construction followed by subset construction and nothing more. The unminimised DFA is already fast enough — one table lookup per character regardless of how many states it has — so the extra pass buys cleanliness we do not need. Minimisation is the standard technique we omit; a production lexer generator that cares about table size would run it.

Putting Thompson's construction and the subset construction together, we have a compilation pipeline from a regular expression to a deterministic state machine: regex parses to syntax tree, syntax tree becomes NFA by induction over its structure, NFA becomes DFA by subset construction. (A textbook pipeline would minimise the DFA here; FRISCCc does not bother, for the reasons just given.) The whole pipeline runs once, when the lexer is built (or, in lex-generated lexers, when the lexer is generated). At run time only the DFA remains, and it costs one character lookup per input character. This is the engineering

miracle of formal-language theory: a notation, a translation, and a data structure that together turn pattern matching into linear-time table lookup.

Key insight

The chain $\text{regex} \rightarrow \text{NFA} \rightarrow \text{DFA}$ is the lexer's backbone. Thompson's construction makes the NFA small; the subset construction makes the DFA deterministic. After both, recognising a string takes constant time per character. (A minimisation pass would make the DFA tight as well; FRISCCc skips it because the unminimised DFA is already fast.) Every lexer generator since `lex` has used some variant of this pipeline, because none of the alternatives has proven faster or simpler in practice.

We have a DFA per token kind. But a real lexer has dozens of token kinds running in parallel against the same input, and they will fight over the boundary between them. Resolving the fight is the central discipline of the practical lexer, and it is what we look at next.

3.4 Maximal munch, and the operator-disambiguation problem

Consider a lexer staring at the start of the input `<= b`. There are two patterns it could match: the operator `<`, of length one, or the operator `<=`, of length two. Both are valid tokens of the language; both are described by regexes in the lexer specification. Which one should the lexer pick?

The answer is the longer match. `<=` is the right tokenisation, because the language does not contain any sensible interpretation of `<` followed immediately by `=` other than the comparison operator. If the lexer picked `<` of length one, it would then have to tokenise `=` of length one, and the parser would see the two-token sequence `< =` where it was expecting `<=`. The parser, having no idea that the two tokens were adjacent in the source, would reject the program. Picking the shorter match would break the language for the sake of an arbitrary scanning order.

This is the principle of *maximal munch*: at every position, take the longest match. It is the default scanning rule of every lexer generator since `lex`, and it is what FRISCCc's lexer does on every byte of input. There is a second rule alongside it, needed when two patterns match the same number of characters: in that case, the rule that appears *earlier in the specification* wins. This is the tie-break, and it is how we declare `int` to be a keyword rather than an identifier — the keyword rule comes first in the specification, so when the lexer is staring at `int x;`, both the keyword regex (matching `int`, length three) and the identifier regex (also matching `int`, length three) accept, and the specification order picks the keyword.

Let us watch the rule in action on a fragment of the chapter's anchor program. [Listing 3.1](#) is the source of `math_fibonacci_iter`, an iterative Fibonacci that we will tokenise piece by piece across this chapter and the next three.

```
1 // EXPECT: 6765
2
3 int main(void) {
4     int n = 20;
5     int a = 0;
6     int b = 1;
7     int i;
8     int t;
9
10    for (i = 0; i < n; i++) {
11        t = a + b;
12        a = b;
13        b = t;
14    }
15
16    return a;
17 }
```

Listing 3.1: The anchor program for [Chapter 3](#) through [Chapter 6](#): an iterative computation of the twentieth Fibonacci number. The function declares five local variables, runs a `for` loop twenty times, and returns the running sum. The expected output — 6765 — is encoded in the comment on line one. The source is seventeen lines long including the comment and blank lines and uses every syntactic feature we care about: declarations, initialised and uninitialised, a `for` statement with all three clauses present, post-increment, compound assignment expressions, and the `return` from a function. Nothing in this program is exciting; that is the point.

Focus on the loop header on line ten: `for (i = 0; i < n; i++) {`. That is a single line of C, and it contains some of the trickiest tokenisation decisions the lexer makes. `for` is a keyword (not an identifier called `for`), because the rule order puts keywords first. The space between `for` and the open parenthesis is whitespace, swallowed silently. The open parenthesis is a one-character token. `i` is an identifier; `=` is an assignment operator, *not* the start of `==`, because the next character (a space) prevents `==` from matching — maximal munch finds one character, not two. The literal `0` is an integer; the semicolon is a punctuator. `i` again, then space, then `<` *not* `<=`, because the next character is a space and the two-character regex `<=` fails to extend. Then `n`, semicolon, `i`, and the sequence `++`.

`++` is the most interesting one. With the lexer at the position of the first `+`, two patterns match: the single-character operator `+`, of length one, and the two-character post-increment `++`, of length two. Maximal munch picks the longer one. The lexer emits the post-increment token and advances by two characters. Figure 3.3 traces the whole sequence character by character.

How does the DFA actually implement the policy? It runs the merged machine (the DFA obtained by taking the alternation of all the per-rule regexes, with each accepting state remembering which rule it came from) against the input, keeping track of the most recent position at which it reached *any* accepting state and which rule was accepted there. When the DFA finally reaches a state with no outgoing transition on the next character — it “dies” — the lexer commits to the most-recently-accepted position: it emits the token of the rule that was accepted there (resolving ties by rule order, since each accepting state knows the index of its winning rule), advances the input pointer to that position, and restarts the DFA from its start state. If no accepting state was ever reached, the lexer reports a lexical error and bails.

The trick of remembering the most-recent accept is the entire policy. Think of a proofreader scanning a passage with a dictionary in hand and a finger over the longest word she has recognised so far. She does not stop at `int` the moment her eye registers a real word; she marks the position and reads on, in case the next few characters extend the match to `intern` or `integer`. Only when the next character cannot extend any word does she commit to the longest word she has seen and start over from there. The scanner does exactly this, with the DFA’s “can I extend?” query replacing the proofreader’s mental dictionary lookup. Without the trailing finger, the DFA would have to know in advance how long a match it was going to find; with it, the DFA gets to be greedy, run until it crashes, and then back up to the latest known-good position. The back-up is by at most a fixed number of characters (the longest match length), so the amortised cost per byte stays constant. Maximal munch adds no asymptotic overhead to the DFA-driven scanner.

Algorithm 3.4 is the entire policy in eighteen lines. The two facts it leans on are that the DFA is deterministic — so the inner loop steps the machine forward without choice — and that acceptance is a property of the current state — so the snapshot in the accept test is a cheap check, not a search. The `lastAccept` marker is what makes the algorithm *maximal*: the scanner does not commit on the first accept it sees; it remembers the position, keeps going, and overwrites the marker on every later accept. When the DFA dies (the inner loop exits because no transition is defined), the most recent snapshot is the longest match, and

for (i = 0; i < n; i++) { — maximal-munch trace

In	Buffer	State	LM	Token emitted	In	Buffer	State	LM	Token emitted
f	f	S1	—		;	;	S12*	21	
o	fo	S2	—		␣	—	dead	21	↔SEMI
r	for	S3*	3		i	i	S5*	23	
␣	—	dead	3	↔KW_FOR	+	—	dead	23	↔ID("i")
(	(	S10*	5		+	+	S8*	25	
i	—	dead	5	↔LPAREN	+	++	S9*	26	
i	i	S5*	7		)	—	dead	26	↔INC(not +)
␣	—	dead	7	↔ID("i")	)	)	S10*	28	
=	=	S11*	9		␣	—	dead	28	↔RPAREN
␣	—	dead	9	↔ASSIGN	{	{	S13*	30	
0	0	S6*	11		eof	—	dead	30	↔LBRACE
;	—	dead	11	↔INT_LIT(0)					
;	;	S12*	13						
␣	—	dead	13	↔SEMI					
i	i	S5*	15						
␣	—	dead	15	↔ID("i")					
<	<	S7*	17						
␣	—	dead	17	↔LT(not <=)					
n	n	S5*	19						
;	—	dead	19	↔ID("n")					

accepting; LM advances (*)
dead: emit ↔rewind ␣ = space (no token)

Figure 3.3: A character-by-character trace of the lexer scanning the fragment `for (i = 0; i < n; i++) {` from line ten of Listing 3.1. Each row corresponds to one byte of input; each column records the lexer’s state immediately after consuming that byte. The lexer maintains a longest-match marker as it scans, advancing it whenever the DFA reaches an accepting state and emitting the corresponding token whenever the DFA dies. The trace makes visible the moments where the decision is non-trivial: `<` commits at the space rather than the digit; `i`, `++`, and the second `)` all force a commit because the DFA cannot extend further. The token stream produced by this fragment is the row of labels along the bottom: keyword, paren, identifier, assign, integer, semicolon, identifier, less-than, identifier, semicolon, identifier, post-increment, paren, brace.

Algorithm 3.4: Maximal-munch token scanner. The scanner drives the DFA forward as long as the next character extends the match, snapshots the position and winning rule on every accepting state it visits, and commits to the most recent snapshot when the DFA dies. The tie-break between rules is already encoded in the DFA: each accepting state carries the smallest rule index among the NFA accept states in its underlying set.

Input: An input buffer *src*, a starting position *p*, and the current mode's DFA *D* with accepting states tagged by rule index.

Output: A token (kind, lexeme, end position) or a lexical error.

```

1 state  $\leftarrow$  start state of D
2 lastAccept  $\leftarrow \perp$  (a position-rule pair, initially undefined)
3 i  $\leftarrow p$ 
4 while i < |src| and  $\delta_D(\textit{state}, \textit{src}[i])$  is defined do
5   | state  $\leftarrow \delta_D(\textit{state}, \textit{src}[i])$ 
6   | i  $\leftarrow i + 1$ 
7   | if state is accepting then
8     | | lastAccept  $\leftarrow (i, \textit{rule}(\textit{state}))$  (remember position and winning rule)
9 if lastAccept =  $\perp$  then
10  | return lexical error at position p
11 (q, r)  $\leftarrow \textit{lastAccept}$ 
12 lexeme  $\leftarrow \textit{src}[p \dots q]$ 
13 execute the actions of rule r (emit, mode switch, line counter)
14 return token for rule r with lexeme lexeme and end q

```

that is the lexeme we commit. The constant-time test on every byte and the single rewind per token are what make maximal munch run in linear time without backtracking.

Maximal munch is not the only possible disambiguation policy, and it has a famous failure case in C++ template syntax. The expression `vector<vector<int>>` was, before C++11, tokenised as `vector`, `<`, `vector`, `<`, `int`, `>>` (right shift), because maximal munch swallowed the two close-angles as a single shift operator. C++98 programmers learned to insert a space: `vector<vector<int> >`. The C++11 standard finally allowed the *parser* to inform the lexer that, in template-argument context, `>>` should be retokenised as two `>`s. FRISCCc has no such problem because its grammar has no nested templates and no context-sensitive operator. Every operator either is or is not maximal-munch reachable from every input position, and the rule order in the specification breaks every tie.

We have been talking about the merged DFA as though it just exists. Building it from a multi-rule specification is a small exercise in extending the constructions we already know. Imagine the spec has k rules, one regex per rule. Combine them into a single regex with alternation: $r_1 \mid r_2 \mid \dots \mid r_k$. Run Thompson on the result. The Thompson NFA has, in its alternation gadget, k outgoing epsilon transitions from the start, one to each $N(r_i)$. Tag each of the k inner accepting states with the index of its rule. Run subset construction. The resulting DFA has accepting states that are sets of NFA states; for each DFA accepting state, the corresponding “winning” rule is the smallest index among the NFA accepting states in the set. That is the tie-break, encoded into the DFA at construction time. At run time, the scanner does not need to know which rules conflicted; each accepting DFA state already carries the right answer.

Key insight

Maximal munch with rule-order tie-break is not just a convention; it is a small piece of code threaded through the DFA construction and the scanning loop. The DFA construction tags each merged accepting state with the winning rule; the scanning loop remembers the most-recent accept and commits there when the machine dies. Together they let one deterministic state machine recognise an entire lexer specification with no backtracking, no lookahead, and a constant amount of work per input character.

We can now chop a stream of characters into tokens, even when several tokens compete for the same starting position. There is one piece of the lexer’s job left, and it is the piece that no amount of maximal-munch cleverness handles by itself: the parts of the source text that are not tokens at all.

3.5 Comments, strings, and the lexer’s modes

Open a C source file at random and you will find regions of text that are not part of the program in any operational sense: line comments starting with `//`, block comments delimited by `/*` and `*/`, and string literals enclosed in double quotes. The lexer has to recognise these regions, consume their contents, and either emit a single token for the whole region (in the case of a string literal) or emit no token at all (in the case of a comment). The question is how the same machinery that handles `++` and `int` handles a block comment such as

```
/* a comment that runs on for paragraphs and contains things
   like int and ++ and tokens that should not be recognised */
```

where the body looks tantalisingly like ordinary source but must not be tokenised at all.

The first instinct is to write a regex for the comment as a whole. For line comments this works fine: `//[^\n]*` matches a double slash followed by anything that is not a newline, which is exactly the language of C line comments. In principle we could plug that regex into the lexer spec, mark the rule’s action as “emit nothing”, and let the merged DFA swallow line comments along with whitespace. Tempting as that is, it is only a hypothetical: FRISCCc does not actually take it. As [Listing 3.2](#) will show, FRISCCc handles even line comments with a dedicated mode, so do not expect to find `//[^\n]*` in the specification file.

Block comments are harder. A regex for them looks like

```
/\[^(*)|\++\[*/\])*\++/
```

which says: open with `/*`, then any number of repetitions of “a non-star, or one or more stars followed by something that is neither a star nor a slash”, then close with one or more stars followed by a slash. It is finite, it is regular, and it is correct; it is also fiddly enough that nobody wants to read it, and a single extra star in the middle will trigger the wrong branch of the alternation and make the lexer reject a legal comment. The regex is *exact* but *unfriendly*.

There is a second instinct, which fails. We could try to write a regex for *nested* block comments — comments that contain comments that contain comments — but no such regex exists. The language of balanced nested delimiters is not regular, as we observed in [Section 3.1](#). C resolves this by simply forbidding nested block comments: when the lexer sees `*/`, the comment ends, even if the opening `/*` introduced a nested comment along the way. The decision was a deliberate one made by the C designers, and it keeps the lexer in the regular fragment.

The third instinct, which works, is to give the lexer *modes*.

A lexer mode is a named state in which the lexer holds a different DFA than in other modes. When the lexer enters a mode, it switches to that mode’s DFA and runs against the input

until something tells it to switch back. The switching is encoded in the rule actions: when the lexer matches `/*`, the action says “enter block-comment mode”; when it matches `*/` from within block-comment mode, the action says “return to initial mode”. Each mode has its own set of rules, its own merged DFA, and its own conventions for what to emit. The lexer as a whole is a tiny state machine *over* modes, with each mode being a regular-language recogniser.

FRISCCc's lexer has four modes, declared at the top of `config/lexer_definition.txt`. The initial mode (the one the lexer starts in) handles all the ordinary tokens: keywords, identifiers, numbers, operators, punctuators, and whitespace. The block-comment mode handles the body of a `/* ... */` comment; the line-comment mode handles the body of a `//` comment up to the next newline; the string mode handles the body of a string literal up to its closing double quote. Switching is one-way descend-and-return: the initial mode enters one of the other three, which eventually return to the initial mode.

```

1  <S_pocetno>//
2  {
3    -
4    UDJI_U_STANJE S_jednolinijskiKomentar
5  }
6  <S_jednolinijskiKomentar>\n
7  {
8    -
9    NOVI_REDAK
10   UDJI_U_STANJE S_pocetno
11 }
```

Listing 3.2: Two mode transitions from `config/lexer_definition.txt`. The first rule says: when the lexer, in the initial mode, sees `//`, emit no token (the dash on the action line) and enter the line-comment mode. The second rule says: when the lexer, in the line-comment mode, sees a newline, emit no token, advance the line counter, and return to the initial mode. The mode names are from the FER PLT specification — `S_pocetno` is “initial state”, `S_jednolinijskiKomentar` is “line-comment state” — and the action keywords `UDJI_U_STANJE` and `NOVI_REDAK` translate to “enter state” and “new line”. Appendix A gives the full glossary. The body of the line comment is consumed by a third rule (not shown) that matches any non-newline character in the line-comment mode and emits nothing.

Listing 3.2 shows the actual transitions. Each rule header tags the mode it applies to (the angle-bracket prefix); each rule body emits a token (or a dash, for nothing), optionally adjusts the line counter for newlines, and optionally switches modes. The lexer engine reads the mode tag, dispatches to that mode's DFA, runs the match, and follows the action.

Lexer modes go by several names in different tools. Flex calls them *start conditions*, declared with `%x` or `%s`; ANTLR calls them *lexer modes*; rust-lexers tend to call them *contexts*. The underlying mechanism is the same: a small non-regular wrapper around a collection of regular sub-machines, with the wrapper’s transitions driven by actions on the regular matches. This is a well-trodden engineering pattern. When the inner language is genuinely a different regular fragment from the outer one — as is the case for the body of a C string literal, which has a completely different escape-character grammar from C source — modes are the clean way to factor it. The alternative, extending the outer DFA’s alphabet to include “in-string” versions of every character, would blow up the DFA’s size for no real gain.

String literals deserve a closer look because their grammar is the busiest of FRISCCc’s auxiliary modes. The mode is entered on a double quote and exited on the next unescaped double quote; in between, the lexer accepts any character that is not a newline or a closing quote, plus the escape sequence `\"` for a literal double quote inside the string. The whole match emits one token, `NIZ_ZNAKOVA` (“character sequence”), carrying the entire lexeme so that the parser and the IR generator can extract the value later. Newlines inside a string trigger a lexical error: FRISCCc, like standard C, does not allow unescaped newlines in string literals.

Modes also explain a couple of things the lexer specification does that look strange at first reading. The rule that opens string mode emits *no* token but executes the action `VRATI_SE 0`, which tells the scanning loop to “go back zero characters” — in effect, “do not consume the opening quote yet, the string-mode rule will match it as part of its own pattern”. This is the FRISCCc lexer’s idiom for letting the mode-switching rule and the next mode’s matching rule coordinate on a shared boundary character. It avoids duplicating the quote-handling logic across two rules. Other lexer generators solve the same problem differently — flex puts the opening quote in the surrounding context’s rule, ANTLR uses a pushed-token mechanism — but the engineering trade-off is the same.

Key insight

Modes turn the lexer from a single DFA into a tiny finite-state machine over DFAs. Each mode is regular; the wrapper that switches between them is not, but it is small enough that we can verify it by inspection. Modes are how the lexer handles regions of source that have their own internal language — comments, string literals, conditional compilation — without giving up the linear-time, constant-memory guarantee of the underlying DFAs.

We now have a theory and a state machine: a notation for patterns (regular expressions), a translation from notation to recogniser (Thompson plus subset construction), a discipline for resolving competing matches (maximal munch with rule-order tie-break), and a wrapper for handling non-token regions (modes). It is time to look at the machinery that actually drives FRISCCc’s lexer.

3.6 What FRISCCc's lexer actually does

The lexer module of FRISCCc lives in `compiler-lexer/` and is, in broad strokes, the implementation of everything we have just described. The interesting parts are the pipeline that turns the specification file into a running scanner, and the contract that the scanner has with the rest of the compiler. We will walk both, taking `math_fibonacci_iter` as the test program at every step.

The pipeline starts with the specification file. Open `config/lexer_definition.txt`. The top of the file declares *macros* — named regex fragments, set off in braces, that other regexes can refer to. `{znak}` is the alphabet of letters, `{znamenka}` is the alphabet of decimal digits, `{bjelina}` is whitespace, and so on. Macros are pure abbreviation; a macro reference in a rule is expanded textually before the regex is parsed. After the macros come two declaration lines: `%X` lists the modes, in the order they will be considered when breaking ties; `%L` lists the token kinds, again in order. After the declarations, the body of the file is a sequence of rules, one per record, each consisting of a mode-tagged regex header, a token kind (or a dash for no token), and zero or more action commands.

The lexer-generation pipeline reads this file and does, in order, the following three things. First, it parses each rule's regex, expanding macros and constructing the syntax tree of the regex. Second, it applies Thompson's construction to each rule's syntax tree, producing an NFA whose accepting state remembers the token kind and the rule's position in the specification. Third, for each mode, it takes the NFAs of all rules tagged with that mode, builds the alternation NFA (introducing a fresh start state with epsilon transitions to each rule's NFA), and runs subset construction to obtain a per-mode DFA. There is no fourth step: FRISCCc does not minimise its DFAs. A textbook generator would run Hopcroft's algorithm here to shrink the tables, but the unminimised DFA already costs one lookup per character, so FRISCCc leaves it alone.

That is the construction phase. It runs once, at lexer initialisation; the result is four DFAs and a small dispatcher that knows which DFA to run in which mode.

The scanning phase is shorter. The lexer maintains a current mode (initially `S_pocetno`) and a current input position. On each iteration of its loop it runs the current mode's DFA against the input starting at the current position, recording the longest-accepting state and the rule that won. When the DFA dies (or runs out of input), the lexer commits to the longest accept: it emits the token of the winning rule (unless the rule emits a dash, in which case it emits nothing), executes the rule's actions (which may switch modes, advance the line counter, or back up the input pointer), and advances the input position past the matched characters. The loop repeats until the input is exhausted, at which point the lexer emits the end-of-input token and returns.

What does the loop produce for our anchor program? Pass [Listing 3.1](#) through `./run.sh --lex` and the lexer writes its output to `compiler-bin/tokens.txt`. [Listing 3.3](#) reproduces the first thirty lines.

```

1  KR_INT      3  int
2  IDN         3  main
3  L_ZAGRADA   3  (
4  KR_VOID     3  void
5  D_ZAGRADA   3  )
6  L_VIT_ZAGRADA 3 {
7  KR_INT      4  int
8  IDN         4  n
9  OP_PRIDRUZI 4  =
10 BROJ        4  20
11 TOCKAZAREZ  4  ;
12 KR_INT      5  int
13 IDN         5  a
14 OP_PRIDRUZI 5  =
15 BROJ        5  0
16 TOCKAZAREZ  5  ;
17 KR_INT      6  int
18 IDN         6  b
19 OP_PRIDRUZI 6  =
20 BROJ        6  1
21 TOCKAZAREZ  6  ;
22 KR_INT      7  int
23 IDN         7  i
24 TOCKAZAREZ  7  ;
25 KR_INT      8  int
26 IDN         8  t
27 TOCKAZAREZ  8  ;
28 KR_FOR      10 for
29 L_ZAGRADA   10 (
30 IDN         10 i

```

Listing 3.3: The first thirty entries of the token stream produced by FRISCcc’s lexer for the source of [Listing 3.1](#). Each entry shows the token kind, the source line, and the lexeme. The complete stream covers all seventeen lines of source and ends in an end-of-input marker; the remainder is in `listings/math_fibonacci_iter/tokens.txt` and the full file appears in Appendix C. Notice how the comment on the first source line generates no tokens at all — it is consumed by the line-comment mode and emitted as nothing — and how the identifier `i` on line ten of the source becomes a single IDN token whose lexeme is the one-character string `i`.

The token stream is exactly the contract the lexer offers to the parser: a finite sequence of tokens, each carrying a kind, a lexeme, a source line and column, and a symbol-table slot for later phases, terminated by an end-of-input marker. The parser can walk it from left to right, with one token of lookahead, and never has to worry about anything we have discussed in this chapter. The parser will not see whitespace; it will not see comments; it will not see strings as anything other than a single character-sequence token with a known lexeme. The compression the lexer performed is total.

There is one more thing the lexer owes the rest of the compiler: useful diagnostics when its job goes wrong. The two interesting failure modes are *illegal characters* (a byte that does not begin any rule's match in the current mode) and *unterminated regions* (end-of-input reached while still in block-comment mode or string mode). FRISCCc reports both with the source line and a short description of what was expected, then sets the compiler's exit code to non-zero and skips the parser entirely. The diagnostic plumbing is the same diagnostic plumbing every other phase uses; we will see it again in [Chapter 4](#).

The lexer's failure modes are not symmetrical with its success modes. A successful tokenisation is a stream of tokens with a known length; a failed tokenisation is one diagnostic and no stream. There is no “partial” output. This is the *fail-fast* discipline we sketched in [Section 2.1](#): a phase either satisfies its contract or refuses to, and downstream phases are entitled to assume the first case if they were invoked at all. It is a tempting optimisation to keep going after a lexical error and try to lex the rest — the user might want to see all errors at once — but the discipline gets harder to maintain quickly, and FRISCCc keeps it simple by stopping at the first one.

The implementation in Java is around nineteen hundred lines spread across the module's eleven classes. `LexerSpecParser` reads the specification file and `LexerGenerator` builds the per-mode DFAs from it (via NFA, DFA, and `NFAtoDFAConverter`); `Lexer`, the largest single file at some six hundred lines, holds the current state and runs the scanning loop; and `Token` is the record that carries kind, lexeme, line, column, and a symbol-table slot. The DFAs themselves are hash-backed transition tables — a `Map` from each state to a `Map` from character to next state, with companion sets for the accepting states and their rule indices. There is no hand-packed integer table here; the entire lexer is those nineteen hundred lines that, between them, implement every construction in this chapter. Anyone reading the source can match each method against the corresponding section above.

What is striking, when we read it side by side with the theory, is how little of the code is theory at all. Most of it is bookkeeping: incrementing the line counter on newlines, looking up the action for a matched rule, formatting diagnostics, dispatching between modes. The DFA simulation itself is a single while loop with about ten lines of body. The theory wrote down the algorithm; the code wrote down everything around it.

Key insight

The lexer is theory plus bookkeeping. The theory — regex to NFA to DFA, plus maximal munch, plus modes — determines what the scanner can do. The bookkeeping — line counters, diagnostics, mode dispatch, action execution — determines whether the scanner is pleasant to use. A correctly engineered lexer leaves the theory running fast in the inner loop and pushes the bookkeeping out to the edges, where it is easy to see and easy to change.

We have tokens. Each one carries a kind, a lexeme, a source line and column, and a slot for the symbol-table index. The stream is finite and ends in a distinguished end-of-input token. The parser of [Chapter 4](#) is entitled to assume every one of those properties, and it is the parser’s job, starting next chapter, to turn the stream into a tree.

Your turn

The companion repository is set up for the first proper build-along of the book. From the companion root, run `cd ch03-lexer` and then `cat README.md`; the README describes the lab and the tinyC subset the chapter targets. The tinyC lexer is smaller than FRISCCc’s — one mode, no string literals, no fixed-point floats, no `const` — so the project fits in a single source file. The starter lives at

```
ch03-lexer/starter/src/main/java/
com/frisccc/tinyC/lexer/Lexer.java
```

with a single TODO marker on the method `tokenize(String)`; your job is to fill it in. The tests live alongside in `ch03-lexer/tests/`, and a working reference implementation is in `ch03-lexer/solution/` for comparison once you are done. Run `mvn test -pl ch03-lexer` from the companion root to execute the test suite. The tests cover every token kind individually plus a handful of small whole-program fragments, including the tinyC version of `math_fibonacci_iter`. When the tests pass, return here for the chapter’s exercises.

Practice

Exercise 3.1 (Conceptual · ★☆☆)

The chapter traces the subset construction on $a(b|c)^*$ and names the ten NFA states $q_0, \dots, q_9$ ([Section 3.3](#)). Without re-reading that derivation, work out $\varepsilon\text{-cl}(\{q_0\})$ for the Thompson NFA produced by the chapter’s inductive gadgets — listing every epsilon-reachable state and explaining which epsilon edge each one enters through. Then compute the two rows of the subset-construction worklist table that correspond to the DFA start state D_0 and the state D_1 reached from D_0 on character `a`: what

NFA subsets do D_0 and D_1 contain, and which of them is an accepting DFA state? (You do not need to complete the full construction; two rows is enough to verify that you understand the worklist algorithm.)

Exercise 3.2 (Applied · ★★★)

FRISCCc's lexer operates in four modes (Section 3.5). Trace the mode machine through the following eighteen-byte source fragment, one token boundary at a time:

```
x = /* hi */ 0;
```

For each segment of the input, record: (a) the mode active when the lexer begins scanning that segment, (b) what the lexer matches, (c) whether a token is emitted or silently discarded, and (d) what mode the lexer is in afterward. At the end, list the complete token stream the lexer hands to the parser. Verify your answer by writing the fragment to a temporary file and running `./run.sh --lex` on it; compare the output in `compiler-bin/tokens.txt` against your prediction.

Exercise 3.3 (Applied · ★★★)

Suppose we tokenise the source string `inta = 5;` with no space between `int` and `a`. Walk through what FRISCCc's lexer does, byte by byte, and predict the token stream it produces. Be precise about the commit point: when the DFA has just consumed the `t` in position 3, it is in a state that is both accepting (for the `int` keyword) and has an outgoing transition (on `a`). Does the lexer emit `int` immediately, or does it keep going? What token does maximal munch ultimately produce for the first lexeme, and why? Check your prediction with `./run.sh --lex` on a file containing exactly `inta = 5;`.

Exercise 3.4 (Applied · ★★★)

Suppose we extend FRISCCc's specification to recognise a new two-character operator `**` for exponentiation, with a token kind `OP_POW`. Write the regex for `OP_POW` (it is a one-liner). Then answer these two questions by reasoning about maximal munch and rule-order tie-breaking (Section 3.4): first, does the new rule need to appear *before* or *after* the existing `ASTERISK` (*) rule in `config/lexer_definition.txt`, and would the lexer behave differently if you swapped their order? Second, give the complete token stream produced for the source text `2 * *3` (with a space between the stars) and for `2**3` (without), and explain why they differ despite using the same two characters.

Exercise 3.5 (Conceptual · ★★☆☆)

The chapter gives the regex for C-style block comments (Section 3.5) and notes that the *nested* variant — comments that can contain `/* ... */` inside themselves — is not regular. Make this claim precise. Describe the language $L_{\text{nest}} = \{\text{well-nested block comments of depth } \geq 1\}$ and apply the pumping lemma for regular languages to prove it is not regular. Your proof should identify the string to pump, the pumping constant p , and the specific pumped string that would be required to be in L_{nest} but is not. Conclude by connecting the result back to the chapter’s engineering decision: why does C forbid nested comments, and what would the lexer need to become if it allowed them?

(This is the chapter’s stretch exercise; expect 30–45 minutes of pencil-and-paper work.)

Project

Exercise 3.6 (Lab · ★★★)

Extend the tinyC lexer in `frisccc-companion/ch03-lexer/` to recognise hexadecimal integer literals. tinyC’s existing integer rule, defined in `tinyc-spec/SPEC.md`, accepts only decimal literals (0 or a non-zero digit followed by more digits). Your goal is to add a new token kind `INT_HEX`, distinct from the existing `INT` kind, and make it match the pattern `0[xX][0-9a-fA-F]+`.

Open the starter file

```
ch03-lexer/starter/src/main/java/
com/frisccc/tinyc/lexer/Lexer.java
```

and find the comment `// TODO: hex literals`. Add the new token kind to the `TokenKind` enum and the new regex rule to the `tokenize` method, positioning it *before* the existing decimal rule. Then open

```
ch03-lexer/tests/src/test/java/
com/frisccc/tinyc/lexer/HexLiteralTest.java
```

The test feeds the source `0xFF + 17` to your lexer and expects the token stream `[INT_HEX(0xFF), PLUS, INT(17)]`. Run

```
mvn test -pl ch03-lexer -Dtest=HexLiteralTest
```

When that passes, run the full suite (`mvn test -pl ch03-lexer`) to confirm you have not regressed any existing behaviour.

Once your code passes, answer these three questions in writing before moving on. First, you were told to place the hex rule before the decimal rule: what would go wrong if you placed it after? Construct a concrete counterexample input to illustrate. Second,

both the hex rule and the decimal rule start with the digit 0; what does maximal munch do when the input is exactly 0 (a lone zero with no following x)? Does it emit INT_HEX, INT, or something else, and why? Third, the INT_HEX token carries the raw lexeme 0xFF — the parser has not yet converted it to an integer value. Which phase of the compiler is the right place to do that conversion, and why is the lexer the wrong place?

Stretch: extend the implementation to accept underscore-separated hex digits (0xFF_00 $\equiv$ 0xFF00), matching the Java 7 and C23 numeric separator syntax. Add a test case for it and update the SPEC.md note on INT_HEX.

Notes and further reading

For lexical analysis the textbook reference is Chapter 3 of the Dragon Book by Aho and collaborators [2], a long and careful walk through regular expressions, NFAs, DFAs, Thompson's construction, subset construction, and the engineering of a complete scanner. The treatment is exhaustive in the way that textbooks are; if you finish this chapter wanting every detail nailed down, the Dragon Book is where to nail them. Cooper and Torczon's *Engineering a Compiler* [18] covers the same material in a more compact and modern style, with better prose and fewer formalisms. Either is a fine companion to this chapter; both treat lex-style scanner generators in detail.

For the historical record, Ken Thompson's 1968 paper [80] is short, lovely, and still worth reading sixty years on. It compiles regex patterns to PDP-7 machine code on the fly — a JIT compiler in the modern sense, half a century before the word was used — and introduces the construction this chapter calls Thompson's. Rabin and Scott's 1959 paper [73] is older still and introduces the subset construction in the broader context of nondeterministic computation; it is the paper that defines the NFA formally and proves it equivalent to the DFA. John Hopcroft's 1971 paper [38] closes the classical pipeline by showing how to find the smallest equivalent DFA in $O(n \log n)$ time; its partition-refinement algorithm is the one that real lexer generators use.

The equivalence of regular expressions, finite automata, and regular languages is due to Stephen Kleene [45] in 1956 — the same paper that named the Kleene star. McNaughton and Yamada in 1960 [61] give an alternative to Thompson's that builds DFAs directly from regexes without going through an NFA; their machines are larger but they skip the subset construction entirely. Brzozowski's derivatives [14], from 1964, are a third alternative and conceptually elegant: a regex's "derivative" with respect to a character is the regex matching the rest of the input, and successive derivatives give a (potentially infinite, but always finitely representable) DFA. Modern lexer libraries occasionally use derivatives; FRISCCc does not, because Thompson plus Rabin–Scott is the path with the most existing tooling and the fewest surprises (Hopcroft's minimisation would normally cap the pipeline, but FRISCCc omits it).

For the underlying theory of formal languages — regular, context-free, context-sensitive, and recursively enumerable — two textbooks dominate. Hopcroft, Motwani, and Ullman’s *Introduction to Automata Theory, Languages, and Computation* [39] is the canonical reference, exhaustive in the way only the third edition of a textbook can be. Michael Sipser’s *Introduction to the Theory of Computation* [75] is shorter, more readable, and the better starting point for a reader who has not seen the material before. Either book will give you the pumping lemma, the Myhill-Nerode theorem, and a proof that nested parentheses are not regular — the three results that fix the boundary between the lexer’s job and the parser’s.

The practical tool that codified Thompson plus subset construction plus minimisation into a usable lexer generator was Mike Lesk’s *lex*, documented in his 1975 Bell Labs technical report [57]. *Lex*’s regex syntax, its rule-action format, its *start conditions* (FRISCcc’s modes), and its tie-break discipline are the conventions every modern lexer generator inherits. Flex, GNU’s free lex, ANTLR’s lexer mode, re2c, and rust-lex all owe their interface to Lesk’s original design. Reading the lex paper is the cheapest way to understand why FRISCcc’s specification file looks the way it does.

Finally, the PLT-FER course on which FRISCcc’s lexer specification is based [27] maintains its lecture notes and project assignments in Croatian; the specification file documented in this chapter uses Croatian state and action names because the course does. Readers who want the Croatian-language treatment of the regular-expression, NFA, DFA, and subset-construction machinery this chapter builds on will find it in Srbljić’s two textbooks [77, 78], which develop the same lex-style scanner pipeline in the notation the course uses. The full glossary, including S_pocetno, UDJI_U_STANJE, NIZ_ZNAKOVA, and the others, lives in Appendix A.

Chapter 4

Reading structure: parsing

“Writing in English is the art of qualifying the previous statement. Writing a grammar is the art of qualifying it before you make it.”
— traditional, often attributed to Donald Knuth

At the end of [Chapter 3](#) we left the chapter’s anchor program as a stream of tokens. The first dozen lines of `compiler-bin/tokens.txt` read `KR_INT`, `IDN(main)`, `L_ZAGRADA`, `KR_VOID`, `D_ZAGRADA`, `L_VIT_ZAGRADA`, `KR_INT`, `IDN(n)`, `OP_PRIDRUZI`, `BROJ(20)`, `TOCKAZAREZ`, and they keep going for another hundred or so atoms until the closing brace of `main`. Every atom carries a kind, a lexeme, and a source line; whitespace and comments are gone. We promised the parser that this stream would be its input. Now the parser has to do something with it.

The problem is that the stream is still flat. Tokens are atoms in the sense that the lexer never has to look beyond one character at a time, but the program they describe is not flat at all. It has nesting — a `for` loop contains a block, which contains expression statements, which contain expressions, which contain more expressions. It has precedence — when we write `a + b * c` inside the loop, the multiplication binds tighter than the addition, and the meaning of the program depends on our knowing that. It has binding — the `else` of an `if-else` attaches to a particular `if`, never to all of them, never to none. The parser’s job is to recover this hidden structure: to take the flat sequence of tokens and reconstruct, out of it, the tree that the programmer had in mind when they typed.

That is what this chapter is about. We are going to spend the first sections building a small theory of context-free grammars, watching where they bind precedence and where they leave it ambiguous, and arguing about which parsing algorithm fits the language we care about. Then we will build, piece by piece, the canonical LR(1) construction that FRISCcc actually uses, watch it grind through the small example `a + b * c`, and finally turn it loose on the anchor program of [Chapter 3](#). By the end we should be able to read the contents of `config/parser_definition.txt` and predict what tree the parser will hand to the semantic analyser of [Chapter 5](#).

4.1 Grammars: the contract between phases

Take the fragment `a + b * c` and ask: what is its structure? A C programmer’s instinct answers “addition, with multiplication on the right”, and draws something like the picture

on the left of Figure 4.1: a tree rooted at the plus, with a on its left and the product $b * c$ on its right. But the tokens on their own do not commit to that picture. A different reader, told only that there are three identifiers and two operators in a row, might just as well have drawn the tree on the right of the figure: a multiplication at the root, with the sum $a + b$ on its left. Both trees have the same yield — the same sequence of leaves, read left to right — and yet they describe two different computations. One of them produces $a + bc$; the other produces $(a + b)c$. The numbers come out different. The programs are not the same.

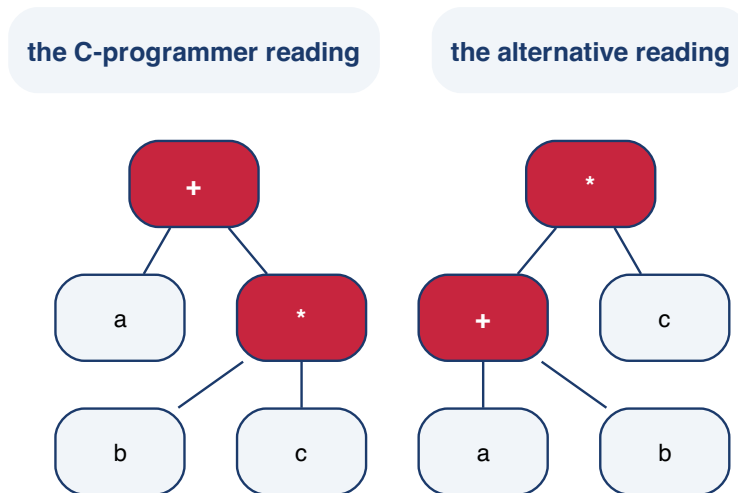


Figure 4.1: Two parse trees with the same yield $a + b * c$. The left tree is what a C programmer expects: the multiplication binds tighter than the addition, so the sum has the product on its right. The right tree is the alternative reading, in which the addition is computed first and then multiplied. Both trees are legitimate context-free-grammar derivations of the same string of tokens; if our grammar admits both, the parser cannot pick between them, and the meaning of the program is undefined. The job of the next few sections is to write a grammar that admits only the left tree.

The parser has to commit to one of the two trees, and it cannot do so by reading the tokens harder. The decision lives in the *grammar* — the document that says, in writing, which sequences of tokens count as legitimate programs and how each sequence is to be parsed. The grammar is the contract between the lexer and everything that comes after it.

A grammar, for our purposes, is the smallest formal object that can say what we need: which sequences of tokens are well-formed, and what tree corresponds to each. The standard

formalism is the *context-free grammar*, written as a collection of rewrite rules. To get a feel for what one looks like, take this three-rule fragment for arithmetic expressions, the one every textbook starts with:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T \cdot F \mid F \\ F &\rightarrow (E) \mid \textit{num} \end{aligned}$$

There are three non-terminals — E , T , F , which we read as “expression”, “term”, “factor” — and five terminals: the plus and dot operators, the two parentheses, and the placeholder *num* standing for a numeric literal. Each rule says “a thing of this kind can be rewritten into one of these alternatives”. A derivation is a sequence of rewrites starting from the start symbol E and ending at a string of terminals. Derive $a + b \cdot c$ (treating each identifier as a *num* for the moment) and the steps tell us exactly which tree the grammar produces:

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T \cdot F \Rightarrow a + F \cdot F \Rightarrow a + b \cdot c$$

Read the derivation as a sequence of decisions. The first rewrite commits to the addition by expanding E into $E + T$. The second chooses to make the left operand of the addition a plain term, the third makes that term a plain factor, and the fourth resolves the factor to the literal a . Then the right operand of the addition gets to expand, and *because the right operand is a T , not an E* , it can be a product. The grammar’s layering — E over T over F — forced the multiplication to live inside the addition’s right child, not above it.

This is the trick we will use throughout the chapter. Encoding precedence in the layering of the grammar means we never have to encode it anywhere else: not in the parser, not in the semantic analyser, not in the IR generator. The shape of the tree carries the information, and every later phase reads it for free.

Formally, a *context-free grammar* is a four-tuple $G = (N, T, P, S)$, where N is a finite set of *non-terminals*, T is a disjoint finite set of *terminals*, $P \subseteq N \times (N \cup T)^*$ is a finite set of *productions* (each written $A \rightarrow \alpha$ with $A \in N$ and α a string of grammar symbols), and $S \in N$ is the distinguished *start symbol*. A single derivation step $\alpha \Rightarrow \beta$ rewrites one non-terminal occurrence inside α by the right-hand side of a production; the reflexive-transitive closure $\Rightarrow^*$ gives the multi-step derivation relation. The *language* $L(G)$ of the grammar is the set $\{w \in T^* \mid S \Rightarrow^* w\}$ of terminal strings derivable from S . A *parse tree* is the data structure of such a derivation: each interior node is a non-terminal, each leaf is a terminal, each node’s children correspond to the right-hand side of the production that expanded it, and the leaves read left to right give the string we started with. A grammar is *ambiguous* if some string in its language has two distinct parse trees.

The Chomsky hierarchy, which we mentioned in passing in [Chapter 3](#), places context-free languages one rung above the regular ones. A regular language is what a finite-state machine can recognise; a context-free language is what a finite-state machine with a stack can recognise. The stack is exactly what we needed to handle balanced parentheses, the example that defeated the lexer’s pumping-lemma argument. Step up one rung on the hierarchy and balanced parentheses become trivial; step up another rung (to context-sensitive grammars) and the parsing problem becomes intractable in general. Context-free is the Goldilocks rung: expressive enough to describe the syntax of every mainstream programming language, weak enough to admit a deterministic parsing algorithm. We are going to spend the rest of the chapter inside it.

4.2 When grammars go wrong: ambiguity

The expression grammar of the previous section was already *stratified*: three layers, one per precedence level, with the layering encoding the binding strength. If we collapse the layers into a single recursive non-terminal,

$$E \rightarrow E + E \mid E \cdot E \mid (E) \mid \textit{num}$$

the grammar becomes prettier and shorter — but also useless. On input $a + b * c$ the new grammar admits both of the trees in [Figure 4.1](#): one derivation expands the outer E into $E + E$ and then the right E into $E \cdot E$; another expands it into $E \cdot E$ and the left E into $E + E$. Both derivations terminate, both yield the same string of terminals, both are legitimate proofs that the string is in the language. Yet they describe different programs. The grammar is ambiguous.

Ambiguity is not just a curiosity. Every phase downstream of the parser walks the parse tree; the shape of the tree is the contract the parser offers them. If the parser can return two different shapes for the same input, the contract has a hole. The semantic analyser would have to choose between them; the IR generator would have to recompute precedence from scratch; the optimisation passes would lose their invariants. The downstream phases are written *against* the tree’s shape, and the tree’s shape is determined by the grammar.

The cure is the layering we already had. Splitting E into the three non-terminals E , T , F does two things at once. It encodes precedence: because the production $E \rightarrow E + T$ has a T on the right, the right operand of a plus can be a product but never another plus at the same level. It encodes associativity: because the recursion in $E \rightarrow E + T$ is on the *left*, a sequence $a + b + c$ parses as $(a + b) + c$, with the leftmost plus deepest in the tree. Two grammatical knobs — which side the recursion goes on, and which non-terminal sits on which side — are enough to specify the shape of every interior expression.

[Figure 4.1](#) would not have been drawable from the stratified grammar. The two trees disagree about which operator is at the root; the layered grammar puts $*$ strictly below $+$ in every parse, by construction. The disambiguation lives in the grammar, not in the parser.

We will see this pattern again when we look at FRISCCc’s full grammar in [Section 4.9](#): a stratified cascade running from assignment, the loosest-binding operator, all the way down to the primary expression, the tightest.

Key insight

A grammar that admits two parse trees for the same input is a grammar that has not made a decision the language needs it to make. The traditional fix is not a smarter parsing algorithm — ambiguity is in the grammar, not in the algorithm — but a more carefully written grammar. Layering by precedence, fixing the recursion side to set associativity, and pushing each operator class down to its own non-terminal are the standard moves. The parser then has only one tree to recover, and the recovery is mechanical.

There is one famous exception to the “fix it in the grammar” rule, the dangling `else`, which we will meet in [Section 4.8](#). C’s syntax really does admit two parses of `if (e) if (f) S else T` and the language standard simply picks one, by convention, at the level of the parser. Every other C ambiguity is fixed in the grammar. The dangling `else` will come back to test whatever algorithm we choose — and we will see in [Section 4.8](#) that LR(1) handles it more cleanly than any top-down alternative could.

4.3 LL or LR? Choosing a strategy

We have a grammar; we need to turn it into a parser. There are two broad strategies, and they differ in which direction they walk the tree. Top-down parsing starts at the root and works its way down to the leaves, expanding non-terminals as it goes. Bottom-up parsing starts at the leaves and works its way up to the root, recognising right-hand sides and reducing them to their left-hand sides. The two strategies have different strengths, and the choice between them shapes everything the rest of the chapter will do.

Top-down is the strategy we would invent if nobody had told us about parsing. Pick a non-terminal, look at the next token, decide which production for that non-terminal starts with that token, and recursively parse each symbol on the production’s right-hand side. This is *recursive descent*, and it is exactly how we would hand-write a parser for a small language. Every non-terminal becomes a procedure; every alternative in a production becomes a branch in that procedure; every terminal becomes a token comparison. The code reads exactly like the grammar; debugging is a matter of stepping through the call stack.

There is a price. Recursive descent is the implementation strategy of $LL(k)$ parsing — “left-to-right input, leftmost derivation, k tokens of lookahead” — and an $LL(k)$ parser can only handle grammars whose productions can be told apart by their first k tokens. The expression grammar of [Section 4.1](#) fails this test the moment we look at it. The production $E \rightarrow E + T$ begins with E , which itself begins with E , which itself begins with E , and so on without bound. There is no k for which the parser can decide, from the first k tokens, whether it is looking at a plain T or an $E + T$. Left recursion is the canonical obstacle

to LL parsing, and removing it mangles the grammar: the standard trick is to rewrite $E \rightarrow E + T \mid T$ as $E \rightarrow TE'$ and $E' \rightarrow +TE' \mid \varepsilon$, which parses correctly but ruins the parse tree's shape and pushes the associativity back into the parser's bookkeeping. Worse, even with left recursion removed, C contains constructions that no fixed lookahead can resolve — the classic example is the declaration `a (b);`, which could be a function call or a declaration of a variable named `b` of type `a`, depending on a typedef the parser has not yet processed. C's grammar is famously not $LL(k)$ for any k .

Bottom-up parsing reverses the picture, and reverses our problem along with it. Instead of guessing what we are about to see, we read tokens, accumulate them on a stack, and *recognise* a right-hand side after we have already collected it. The parser shifts each terminal onto a stack, watches the top of the stack match the right-hand side of some production, and reduces that right-hand side to its left-hand side. The decision is made not at the start of the construct but at the end of it, when all the evidence is in. Bottom-up parsing handles left recursion natively; it can handle most of C's grammar with one token of lookahead; and it admits a uniform, table-driven implementation that does not need to be rewritten when the grammar changes.

The largest class of context-free grammars that admits a deterministic parser of any kind is the class of $LR(k)$ grammars for some fixed k . Donald Knuth proved this in his 1965 paper *On the translation of languages from left to right* [48]. The bound is tight in a strong sense: every deterministic context-free language is $LR(k)$ for some k , and every $LR(k)$ grammar can be parsed deterministically by a table-driven shift-reduce automaton. The construction with $k = 1$ is expressive enough to cover the syntax of every mainstream imperative language, and it is the construction we will spend the rest of this chapter building. Knuth's paper is short, lovely, and notoriously the first place anyone learning parser theory should start.

The analogy that makes shift-reduce parsing click is the *stack machine*. Every bottom-up parser is a finite-state machine wrapped around a stack; the states encode “what have we seen so far”, the stack holds the partial parse trees we have already recognised, and the transitions are guided by the next token of lookahead. This is exactly the picture that comes up later in the book when we look at the FRISC ISA — a stack-machine VM is how Chapter 12 will run bytecode — and it is the same picture that shows up in postfix expression evaluation, in proof checking, and in every other context where partial information accumulates left to right. Naming the analogy is half the battle.

We are going to use $LR(1)$. The remaining sections build the construction one piece at a time.

4.4 LR(1) items: snapshots of a parse in progress

Imagine the parser is partway through reading the string $(a + b)$. It has consumed the open paren, the identifier a , and the plus, and the next token is the identifier b . What does the parser know at this moment?

Whatever it knows is some answer to the question “which production is the parser working on, and how far along is it?”. The parser cannot say with certainty that it is recognising $E \rightarrow E + T$; the addition has not finished yet, and the right operand might turn out to involve a multiplication. But it can say something close: “I have already matched an E and a plus; if the next part matches a T , I will commit to $E \rightarrow E + T$ and reduce”. The position of the parser inside the production is the piece of state we need to record, and the standard way to record it is called an *LR(0) item*: a production with a dot somewhere inside its right-hand side. We write the item by putting a literal dot at the position the parser has reached:

$$E \rightarrow E + \cdot T$$

reads as “we have matched E followed by $+$, and we are about to match T ”. The dot is the parser’s view of how much of the production has gone by, as Figure 4.2 illustrates with three snapshots of a reading head sliding left to right across the right-hand side. An item with the dot at the very right — $E \rightarrow E + T \cdot$ — is *complete*, and tells us the parser is ready to reduce; an item with the dot earlier is *partial*, and tells us the parser is still collecting evidence.

This is the LR(0) item. The “1” in LR(1) adds one more piece of information: a *lookahead* terminal, the token the parser expects to see *after* the production has been reduced. The lookahead is what buys LR(1) its precision; when several productions are simultaneously possible at a state, the lookahead lets us tell them apart even when their left contexts agree.

An *LR(0) item* is a production $A \rightarrow \alpha\beta$ with a dot inserted at some position in the right-hand side, written $A \rightarrow \alpha \cdot \beta$. The dot records how far the parser has progressed. An *LR(1) item* pairs an LR(0) item with a *lookahead* terminal a : the pair $[A \rightarrow \alpha \cdot \beta, a]$ means the parser is partway through recognising $A \rightarrow \alpha\beta$ and expects to see a on the input immediately after the reduction completes. The first component records position in the production; the second records the right context expected once the position reaches the end.

To see the lookahead in action, consider the augmented grammar $S' \rightarrow E$ for the expression sub-language, with the artificial start symbol S' added at the top so we have a unique production whose reduction signals acceptance. The very first item of the parse is

$$[S' \rightarrow \cdot E, \$]$$

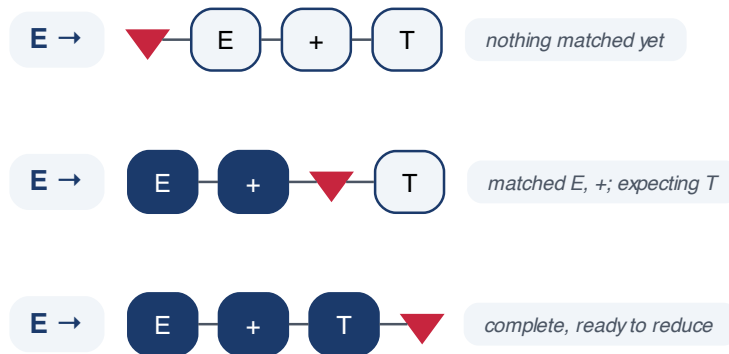


Figure 4.2: Three snapshots of the parser inside the production $E \rightarrow E + T$, drawn as positions of a reading head moving left to right across the right-hand side. At the start (top) the parser has matched nothing; in the middle, the parser has matched E and $+$ and is poised on T ; at the bottom, the parser has matched the whole right-hand side and is ready to reduce. The dot in the textual notation $E \rightarrow E + \cdot T$ is exactly the position of the head.

where $\$$ is the conventional end-of-input marker. The parser has matched nothing, is about to match E , and expects $\$$ when the match completes. This item generates a whole family of items, by the closure operation we will see in the next section, and those items together describe the initial state of the parser.

The intuition is worth one more rephrasing. Each LR(1) item is a *hypothesis* the parser is entertaining about what it is recognising. The set of items at any state is the set of all hypotheses the parser thinks are simultaneously plausible. When the parser shifts a token, all hypotheses advance their dots; when none of them advance, the parser is stuck (or has a complete hypothesis to reduce). The mechanics are simple; the bookkeeping is what takes the next two sections.

4.5 Closure and goto: building the item sets

Start from the initial item $[S' \rightarrow \cdot E, \$]$. The parser has not consumed anything; its only hypothesis is that whatever comes next is going to be an E . But E is a non-terminal, and a non-terminal does not appear in the input directly — it can only appear as the left-hand side of a production. So if the parser is about to recognise an E , it is in principle also about to recognise the right-hand side of *some* production for E . Looking at our tiny grammar, those productions are $E \rightarrow E + T$ and $E \rightarrow T$. The parser is therefore equally entertaining the hypotheses

$$[E \rightarrow \cdot E + T, ?] \quad \text{and} \quad [E \rightarrow \cdot T, ?]$$

with the dot at the start of each. We will fill in the question marks in a moment; for now, notice that the same argument cascades: each new item ends in something that begins with a non-terminal, which means more items get added in turn, which means more non-terminals appear, and the set keeps growing until we run out of new items. The cascade is the *closure* operation, and it is what turns a single seed item into the full set of hypotheses the parser holds at a state.

The closure of an item set I is defined inductively. Start with I . For every item $[A \rightarrow \alpha \cdot B\beta, a]$ in the current set, and for every production $B \rightarrow \gamma$ of the grammar, and for every terminal $b \in \text{First}(\beta a)$ (the set of terminals that can begin a string derivable from βa , see the fixpoint computation in any standard treatment such as the Dragon Book [2]), add the new item $[B \rightarrow \cdot \gamma, b]$ to the set. Repeat until no new items appear. The result, written $\text{closure}(I)$, is the smallest set containing I and closed under this rule. Because the underlying set of LR(1) items is finite — bounded by $|P|$ times the maximum right-hand-side length plus one, times $|T| + 1$ — the iteration terminates. The full implementation lives at `compiler-parser/.../lr/LRClosure.java`, where the inductive rule above is a single while loop that re-scans the item set until a full pass adds nothing new — a naive fixpoint equivalent to the worklist form we

present here for clarity.

Algorithm 4.1: LR(1) item-set closure: expand every dotted item whose dot stands before a non-terminal by adding seed items for each of that non-terminal’s productions, with lookaheads drawn from $\text{First}(\beta a)$.

Input: LR(1) item set I ; grammar $G = (N, T, P, S)$.

Output: The closure $\text{closure}(I)$: the smallest superset of I closed under the closure rule.

```

1  $C \leftarrow I$ ;
2  $W \leftarrow$  a worklist initialised to  $I$ ;
3 while  $W$  is non-empty do
4   pop an item  $[A \rightarrow \alpha \cdot B\beta, a]$  from  $W$ ;
5   if the symbol immediately after the dot is a non-terminal  $B$  then
6     for each production  $B \rightarrow \gamma$  in  $P$  do
7       for each terminal  $b \in \text{First}(\beta a)$  do
8         if  $[B \rightarrow \cdot \gamma, b] \notin C$  then
9           add  $[B \rightarrow \cdot \gamma, b]$  to  $C$ ;
10          push  $[B \rightarrow \cdot \gamma, b]$  onto  $W$ ;
11 return  $C$ ;
```

Lookaheads propagate through closure by a rule that is worth staring at for a moment. When we expand $[A \rightarrow \alpha \cdot B\beta, a]$ by a production $B \rightarrow \gamma$, the new item is $[B \rightarrow \cdot \gamma, b]$ for each $b \in \text{First}(\beta a)$. The logic is: once we finish recognising B , we will be inside the original A production, immediately before β . So the token we expect to see after the B reduction is whatever can begin β — or, if β can be empty, whatever can begin the original lookahead a . The first-set computation handles the case where β contains non-terminals that themselves can derive empty strings; the FRISCCc grammar has no epsilon productions, so in our case $\text{First}(\beta a)$ is always $\text{First}(\beta)$ when β is non-empty.

Run the closure on the singleton $\{[S' \rightarrow \cdot E, \$]\}$ and the result is the initial item set I_0 . For the three-rule expression grammar of [Section 4.1](#), the closure has seven items:

```

 $[S' \rightarrow \cdot E, \$]$ 
 $[E \rightarrow \cdot E + T, \$ / +]$ 
 $[E \rightarrow \cdot T, \$ / +]$ 
 $[T \rightarrow \cdot T \cdot F, \$ / + / \cdot]$ 
 $[T \rightarrow \cdot F, \$ / + / \cdot]$ 
 $[F \rightarrow \cdot (E), \$ / + / \cdot]$ 
 $[F \rightarrow \cdot \text{num}, \$ / + / \cdot]$ 
```

where each item carries the set of lookaheads it inherited from its ancestors. We have written multiple lookaheads on a single line for compactness; formally these are several LR(1) items sharing the same dotted production but differing in their lookahead.

The second operation we need is *goto*. Suppose the parser is in state I and shifts symbol X . Which items survive? The ones whose dot was just before X , advanced to be just after it. We collect those advanced items, close the resulting set, and call the result $\text{goto}(I, X)$. Algorithmically:

For an item set I and a grammar symbol $X \in N \cup T$, define $\text{goto}(I, X) = \text{closure}(\{[A \rightarrow \alpha X \cdot \beta, a] \mid [A \rightarrow \alpha \cdot X \beta, a] \in I\})$. The goto operation takes all items in I whose dot is just before X , advances the dot one position past X , and closes the resulting set. If no items in I have X immediately after the dot, the goto is empty. The implementation in `compiler-parser/.../lr/LRGoto.java` mirrors this definition directly: filter, advance, close.

Algorithm 4.2: LR(1) GOTO transition: filter the items of I whose dot stands immediately before X , advance each of their dots by one position, then close the resulting kernel using Algorithm 4.1.

Input: LR(1) item set I ; grammar symbol $X \in N \cup T$; grammar G .

Output: $\text{goto}(I, X)$: the item set reached from I by shifting X and closing the result.

```

1  $J \leftarrow \emptyset$ ;
2 for each item  $[A \rightarrow \alpha \cdot X \beta, a] \in I$  do
3    $\mid$  add  $[A \rightarrow \alpha X \cdot \beta, a]$  to  $J$ ;
4 return  $\text{closure}(J)$ ;
```

Take the closure I_0 we just computed. From I_0 , on symbol E , the goto consists of $[S' \rightarrow E \cdot, \$]$ and $[E \rightarrow E \cdot + T, \$ / +]$, both with dot just past the E , plus whatever closure adds. Closure adds nothing here because neither item has a non-terminal immediately after the dot, so $\text{goto}(I_0, E)$ is exactly those two items. Call the result I_1 . Similarly, $\text{goto}(I_0, T)$ contains $[E \rightarrow T \cdot, \$ / +]$ and $[T \rightarrow T \cdot \cdot F, \$ / + / \cdot]$ — the latter with the $\cdot$ operator just after the dot, ready to start recognising a multiplication. Call this set I_2 . And so on for the other symbols.

Algorithm 4.3 is the bookkeeping that turns one closure into many. The outer worklist treats each item set as a node and each goto destination as an edge, and the algorithm halts because the set of possible item sets is finite (every item is bounded by the grammar size, and lookaheads are drawn from a finite set of terminals). What falls out is the entire LR(1) automaton — nodes are item sets, edges are gotos — and the action table of the next section reads straight off it. Figure 4.3 draws the first six states the algorithm produces on the three-rule expression grammar; follow the edges and you are watching the worklist execute.

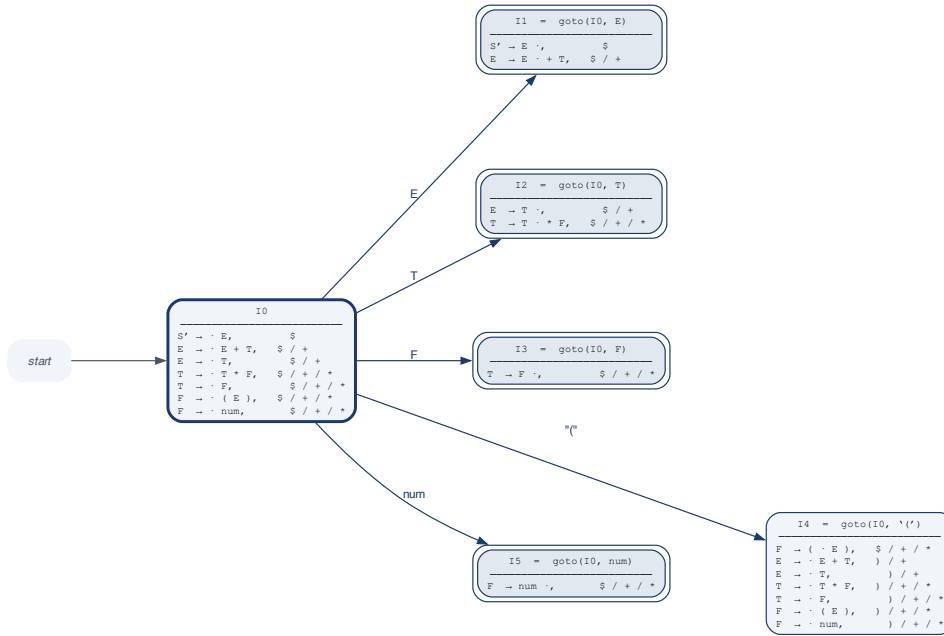


Figure 4.3: The LR(1) automaton for the three-rule expression grammar, restricted to the first six states. The start state I_0 is the closure of the seed item; arrows out of I_0 are labelled by the grammar symbol on which the goto is taken, and lead to new item sets I_1 through I_5 . States with at least one complete item (dot at the rightmost position) are shaded; from such states the parser will reduce on the appropriate lookahead. The full automaton has more states — the closure cascade produces a few more sets reachable by shifting through parentheses and combining plus and dot operators — but the fragment shown here is enough to see the structure: each state collects every hypothesis the parser is willing to entertain at that point, and each transition advances those hypotheses by exactly one symbol.

Algorithm 4.3: Canonical collection of LR(1) item sets: starting from the closure of the augmented seed item, repeatedly apply Algorithm 4.2 on every grammar symbol, adding any newly-discovered set to a worklist, until no new sets appear. The result is the state set of the LR(1) automaton together with its transition graph.

Input: Grammar G augmented with $S' \rightarrow S$ and end marker $\$$.

Output: The canonical collection $\mathcal{C} = \{I_0, I_1, \dots\}$ of LR(1) item sets and the transitions among them.

```

1  $I_0 \leftarrow \text{closure}(\{[S' \rightarrow \cdot S, \$]\});$ 
2  $\mathcal{C} \leftarrow \{I_0\}; W \leftarrow \text{worklist containing } I_0;$ 
3 while  $W$  is non-empty do
4   pop an item set  $I$  from  $W$ ;
5   for each grammar symbol  $X \in N \cup T$  do
6      $J \leftarrow \text{goto}(I, X);$ 
7     if  $J$  is non-empty then
8       if  $J \notin \mathcal{C}$  then
9         add  $J$  to  $\mathcal{C}$  and push  $J$  onto  $W$ ;
10      record the transition  $I \xrightarrow{X} J$ ;
11 return  $\mathcal{C}$  and its transitions;
```

Repeat the construction. Start from I_0 , take goto on every grammar symbol, add the new sets to a worklist, and keep going until no new sets appear. The result is the *canonical collection* of LR(1) item sets, which is the set of states of the parser we are building. For the three-rule expression grammar the collection has on the order of a dozen states; for the FRISCCc grammar it has on the order of a thousand. Either way the construction is mechanical: nothing in it requires cleverness, only patience and a worklist.

Figure 4.3 shows the first six states of the canonical collection for our running example, drawn as the transition graph of an automaton. The shape is striking. Each state is a snapshot of the parser's plausible hypotheses; each edge is a step the parser can take, labelled by the symbol that triggers it. The automaton is finite (because the canonical collection is finite), deterministic (because the goto function is single-valued), and small enough to draw on the page. It is also, importantly, the last piece of the construction we have to do by hand. From here on, the ACTION and GOTO tables fall out of the automaton mechanically.

4.6 ACTION and GOTO: the parser's table

The canonical collection of LR(1) item sets is the parser's state-transition graph, but it is not the parser's data structure. What the parser runs against is a pair of *tables* that together encode every transition and every reduction. We call them ACTION and GOTO: the

first holds the parser's response to a terminal, the second its response to a non-terminal after a reduction.

The reasoning is short. At each state, exactly three kinds of decision can come up. If the parser sees a terminal a and the current state has at least one item whose dot is just before a , the parser *shifts* a onto the stack, transitions to the state $\text{goto}(I, a)$, and reads the next token. If the parser sees a terminal a and the current state has a complete item $[A \rightarrow \alpha \cdot, a]$ — complete, with a as a lookahead — the parser *reduces* by the production $A \rightarrow \alpha$, pops $|\alpha|$ items off the stack, and uses the GOTO table to find the state to push back. If the parser sees the end-of-input marker $\$$ in the final state $[S' \rightarrow E \cdot, \$]$, the parser *accepts*, and the parse is finished. Every other entry in the ACTION table is an *error*: there is no item that can advance on this token, and the input is rejected.

Take state I_2 from our running example, the one containing $[E \rightarrow T \cdot, \$/+]$ and $[T \rightarrow T \cdot \cdot F, \$ / + / \cdot]$. The first item is complete and reduces on lookahead $\$$ or $+$. The second is partial and shifts on $\cdot$. So the ACTION row for I_2 reduces by $E \rightarrow T$ in the columns for $\$$ and $+$, shifts in the column for $\cdot$, and — because the very same state is also entered from inside a parenthesised sub-expression, a wrinkle we take up alongside the table below — reduces on $)$ as well; every remaining column is an error. That single row encodes the parser's behaviour at I_2 for every possible next token, in constant space and constant lookup time.

Algorithm 4.4: ACTION and GOTO table construction from the canonical collection. Each item in each state contributes at most one entry; cells that receive two distinct entries are conflicts and must be resolved (by tie-break for shift–reduce, by grammar edit for reduce–reduce).

Input: Canonical collection $\mathcal{C} = \{I_0, \dots, I_{n-1}\}$ from Algorithm 4.3; numbered productions of G .

Output: Tables $\text{Action}[\cdot, \cdot]$ and $\text{Goto}[\cdot, \cdot]$ for the LR(1) parser.

```

1 initialise every Action cell to error and every Goto cell to undefined;
2 for each item set  $I_i \in \mathcal{C}$  do
3   for each item  $[A \rightarrow \alpha \cdot \beta, a] \in I_i$  do
4     if  $\beta = X\gamma$  for some terminal  $X$  and  $\text{goto}(I_i, X) = I_j$  then
5       set  $\text{Action}[i, X] \leftarrow \text{shift } j$ ;
6     else if  $\beta = \varepsilon$  and  $A \rightarrow \alpha$  is production  $k$  and  $A \neq S'$  then
7       set  $\text{Action}[i, a] \leftarrow \text{reduce } k$ ;
8     else if  $[S' \rightarrow S \cdot, \$] \in I_i$  then
9       set  $\text{Action}[i, \$] \leftarrow \text{accept}$ ;
10  for each non-terminal  $A \in N$  with  $\text{goto}(I_i, A) = I_j$  do
11    set  $\text{Goto}[i, A] \leftarrow j$ ;
12 report any cell that received two distinct actions as a conflict;
13 return Action and Goto;
```

[Algorithm 4.4](#) reads the table off the automaton in a single sweep: shift entries fall out of the transitions on terminals, goto entries fall out of the transitions on non-terminals, reduce entries are dictated by the complete items inside each state, and the accept entry is the lone item $[S' \rightarrow S \cdot, \$]$ in the start state's goto on S . Two items in the same state asking for two different actions on the same lookahead is a conflict, and a conflict is the algorithm's way of telling us that the grammar is not LR(1). The result, when the grammar cooperates, is a single matrix whose every cell tells the parser what to do, in constant time, at runtime. Before we read the matrix off, let us fix the production numbers once and for all, because the reduce entries name productions by number:

- | | |
|-------------------------------|--------------------------------|
| (1) $E \rightarrow E + T$ | (2) $E \rightarrow T$ |
| (3) $T \rightarrow T \cdot F$ | (4) $T \rightarrow F$ |
| (5) $F \rightarrow (E)$ | (6) $F \rightarrow \text{num}$ |

We write multiplication as $\cdot$ in the grammar and item sets and as $*$ in the table and code; they denote the same operator. A cell that reads r4 means “reduce by production 4, that is $T \rightarrow F$ ”; a cell that reads s7 means “shift the lookahead and go to state I_7 ”. The two integers have nothing to do with each other — one indexes a production, the other a state — even though both happen to be small. [Table 4.1](#) is the *complete* table the algorithm produces on the three-rule expression grammar: all twelve states, the six the automaton figure drew and six more that carry the parser through parentheses.

This is the whole table, and every state that any entry names — I_6 , I_7 , I_8 , and the rest — now has a row of its own, so a trace can be followed end to end without ever pointing off the page. Even for the FRISCcc grammar, with its thousand-odd states and forty-odd terminals, the table is only a few hundred kilobytes: a one-time construction cost paid by LRTableBuilder, after which the parser runs at the speed of a table lookup per token.

One detail repays a second look. When we built the item sets by closure, each set carried the lookaheads of the path that reached it, and the path from I_0 never put a right parenthesis into the lookahead of I_2 , I_3 , or I_5 . Yet those three rows reduce on $)$ in the finished table. The reason is that each of them is reached by more than one path: I_2 , for instance, is the state the parser enters after reducing a term at the top level (where the next token is $\$$ or $+$) *and* after reducing a term inside a parenthesised sub-expression (where the next token can be $)$). The finished table keeps one row per state no matter how the parser arrived, so each row's reduce action covers the union of the lookaheads from every arrival path. This pooling is the compaction the Notes take up at the end of the chapter under the name LALR(1). We show the twelve-row pooled form here because it fits on a page and accepts exactly the same programs, building exactly the same trees; FRISCcc's own table-builder keeps the canonical states apart instead, which is why its tables run larger. For the strings we trace the difference is invisible, because a right parenthesis only ever shows up as a lookahead when we genuinely are inside parentheses.

state	ACTION						GOTO		
	num	(	)	+	*	\$	<i>E</i>	<i>T</i>	<i>F</i>
I_0	s5	s4					1	2	3
I_1				s6		acc			
I_2			r2	r2	s7	r2			
I_3			r4	r4	r4	r4			
I_4	s5	s4					8	2	3
I_5			r6	r6	r6	r6			
I_6	s5	s4						9	3
I_7	s5	s4							10
I_8			s11	s6					
I_9			r1	r1	s7	r1			
I_{10}			r3	r3	r3	r3			
I_{11}			r5	r5	r5	r5			

Table 4.1: The complete ACTION/GOTO table for the three-rule expression grammar, all twelve states. Each entry sk means “shift the lookahead and transition to state I_k ”; each entry rk means “reduce by production k ” in the numbering fixed just above ($1: E \rightarrow E + T$, $2: E \rightarrow T$, $3: T \rightarrow T \cdot F$, $4: T \rightarrow F$, $5: F \rightarrow (E)$, $6: F \rightarrow num$); acc is the accepting action; a blank cell in the ACTION block is an error. The three rightmost columns are the GOTO table, consulted after a reduction to pick the next state from the non-terminal just produced; a blank GOTO cell is never consulted, because the parser never reduces to that non-terminal from that state. States I_6 through I_{11} are the ones the automaton figure (Figure 4.3) left out: I_6 and I_7 open the right operand of a $+$ and a $\cdot$, while I_8 , I_9 , I_{10} , and I_{11} walk the parser through a parenthesised sub-expression.

The crucial property of the ACTION table is that every entry is single-valued. If we built the canonical collection of LR(1) item sets and the resulting ACTION table has at most one of shift, reduce, or accept in every cell, the grammar is *LR(1)* and the parser is deterministic. If some cell holds more than one action, we have a *conflict*, and the grammar is not LR(1) as written. The conflict is the algorithm's way of telling us where the grammar is ambiguous or where the lookahead is not enough.

Theorem (Determinism of LR(1) parsing). *If the canonical ACTION table for a grammar G contains at most one action per cell, then for every input either the LR(1) parser accepts it (producing a unique parse tree) or rejects it (at a uniquely determined position). The parser makes no choices at runtime; the action at each step is determined by the current state and the lookahead.*

The proof is a short induction on the length of the run, using the single-valued action property as the inductive step. The full version is in Theorem G.16 of Appendix G, and the textbook treatments in Aho et al. [2] and Grune and Jacobs [33] carry the standard detailed argument. A useful corollary, also classical: every LR(1) grammar is unambiguous. The converse fails — there are unambiguous grammars not in LR(1) — but the LR(1) class is large enough to cover every mainstream programming language.

The determinism theorem is the entire engineering case for LR(1). We get a parser whose every step is forced; we get a static check that the grammar is unambiguous (because conflicts would show up in the table); and we get the algorithm in a form that runs as a table-driven loop, with no recursion and no backtracking. The cost is the table itself, which we pay once at startup and then forget about.

4.7 Tracing a parse on $a + b * c$

We have a table; we need to watch it run. The driver of an LR(1) parser is short. We maintain a stack of states (only the states, not the symbols — the parse tree is built incrementally on the side as reductions fire). We read the next token from the input. We look up ACTION[top of stack, lookahead] in the table. If the entry says shift, we push the destination state and read the next token; if it says reduce by $A \rightarrow \alpha$, we pop $|\alpha|$ states, look up GOTO[new top, A], push that, and emit a tree node attaching the popped children under a fresh A ; if it says accept, we stop; if it says error, we diagnose and recover. That is the whole algorithm, set out as Algorithm 4.5.

Let us run it on $a + b * c$, treating each identifier as a *num* to match the table of Table 4.1. Table 4.2 traces the parse step by step. Read the stack column as the parser's complete state, the input column as what is left to read, and the action column as the entry from the ACTION table that fired. After every reduction the parser attaches the popped subtrees as children of a fresh node and the tree grows by one interior node from the bottom up. After fourteen steps the parser reaches the accept action, and the tree is complete: an E at the

Algorithm 4.5: Shift–reduce LR(1) parsing driver. The state stack carries automaton states; the tree stack accumulates partial parse trees that are combined whenever a reduction fires. Each iteration consults a single Action cell and either pushes, pops and pushes via Goto, accepts, or reports an error.

Input: Token stream $w\$$; tables Action and Goto from Algorithm 4.4.

Output: Either a parse tree for w or a syntax error.

```

1 initialise the state stack  $S \leftarrow \langle 0 \rangle$ ;
2 initialise the tree stack  $\mathcal{T} \leftarrow \langle \rangle$ ;
3  $a \leftarrow$  the first token of  $w\$$ ;
4 while true do
5   let  $s$  be the top of  $S$ ;
6   switch Action[ $s, a$ ] do
7     case shift  $j$  do
8       push  $j$  onto  $S$ ; push a leaf node for  $a$  onto  $\mathcal{T}$ ;
9        $a \leftarrow$  the next token;
10    case reduce  $A \rightarrow \beta$  do
11      pop  $|\beta|$  entries from  $S$  and  $|\beta|$  subtrees from  $\mathcal{T}$ ;
12      let  $s'$  be the new top of  $S$ ; push Goto[ $s', A$ ] onto  $S$ ;
13      attach the popped subtrees as children of a fresh  $A$ -node and push it onto  $\mathcal{T}$ ;
14    case accept do
15      return the single tree in  $\mathcal{T}$ ;
16    case error do
17      report a syntax error at  $a$  and recover;

```

root, with $E + T$ beneath it, with T on the right expanding into $T \cdot F$, and so on down to the three leaves a, b, c .

step	stack	input	action	note
1	$\emptyset$	$a + b * c \$$	shift 5	read a
2	$\emptyset 5$	$+ b * c \$$	reduce 6	$F \rightarrow num$
3	$\emptyset 3$	$+ b * c \$$	reduce 4	$T \rightarrow F$
4	$\emptyset 2$	$+ b * c \$$	reduce 2	$E \rightarrow T$
5	$\emptyset 1$	$+ b * c \$$	shift 6	read $+$
6	$\emptyset 1 6$	$b * c \$$	shift 5	read b
7	$\emptyset 1 6 5$	$* c \$$	reduce 6	$F \rightarrow num$
8	$\emptyset 1 6 3$	$* c \$$	reduce 4	$T \rightarrow F$
9	$\emptyset 1 6 9$	$* c \$$	shift 7	read $*$
10	$\emptyset 1 6 9 7$	$c \$$	shift 5	read c
11	$\emptyset 1 6 9 7 5$	$\$$	reduce 6	$F \rightarrow num$
12	$\emptyset 1 6 9 7 10$	$\$$	reduce 3	$T \rightarrow T \cdot F$
13	$\emptyset 1 6 9$	$\$$	reduce 1	$E \rightarrow E + T$
14	$\emptyset 1$	$\$$	accept	tree complete

Table 4.2: Shift–reduce trace for the input $a + b * c$ against the grammar of Section 4.1 and the table of Table 4.1. Each row records the parser’s state stack, the remaining input, and the action the table dictated at that step. Reductions name the production by its number from the legend of Table 4.1. After step 13 the parser has built the full parse tree with E at the root, $E + T$ beneath it, and the multiplication living inside the right T . Every state the trace visits — including 9 and 10, reached by goto after the inner reductions — has its own row in Table 4.1, so each action below can be checked against the table directly.

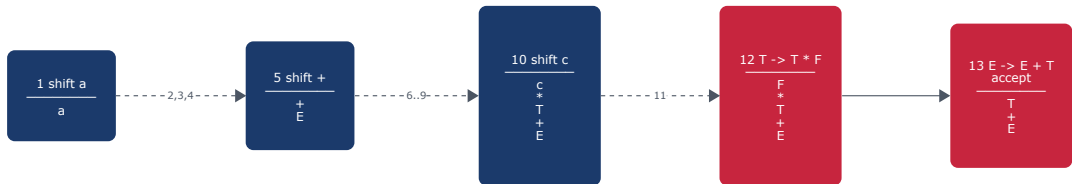


Figure 4.4: Five snapshots of the LR(1) symbol stack during the parse of $a + b * c$. The figure visualises the conceptual symbol stack — the grammar symbols the automaton states stand for — as a teaching aid; the driver itself stores only the states. Navy cards mark a shift (the top symbol was just pushed); red cards mark a pending reduction (the shaded symbols are about to be popped and replaced by a single non-terminal). Card height is proportional to stack depth: step 10 is the peak at five symbols, after which three successive reductions (steps 11–13) collapse the stack back down toward the single root E . Steps elided between snapshots are noted on the dashed arrows; the full trace is in Table 4.2.

Figure 4.4 draws the same parse five snapshots at a time. The picture’s shape tells the same story the table does, only faster: the stack grows on every shift and collapses on every reduction, and the deepest peak is the moment just before the multiplication’s right-hand side falls together.

The trace makes one feature of LR(1) parsing concrete. The parser never guesses. At every row, the action is determined uniquely by the top of the stack and the lookahead — the same two pieces of information the table is indexed by. There is no look-ahead beyond the current token; there is no try-and-backtrack; there is no ambiguity resolution at runtime. The construction did the hard thinking once, and the runtime does table lookups forever after.

The shape of the tree the parse builds is exactly the shape the grammar dictated. The first plus to be reduced is the addition, because the multiplication on its right has not finished accumulating evidence at the time of the plus’s reduction — and indeed the parser shifts past the plus, then walks into the right-hand-side T that eventually expands into $T \cdot F$, then reduces the multiplication, then reduces the plus over the multiplication’s result. The associativity of the addition came for free; the precedence of the multiplication came for free; and the tree we get is the tree on the left of Figure 4.1, not the one on the right.

Key insight

The LR(1) parser is a stack machine driven by a table. The table encodes every parsing decision the grammar permits, indexed by the current state and one token of lookahead. At runtime the parser does nothing but consult the table, push or pop states, and attach tree nodes to a growing parse tree. The whole loop is fewer than twenty lines of code (the driver in `LRParser.java` [27]) and runs in time linear in the input.

4.8 Conflicts, and what they tell you

So far the construction has been clean: the canonical collection yields a single-valued ACTION table, the table yields a deterministic parser, the parser builds the tree. In practice the construction sometimes *fails* to be single-valued, and the table-builder reports a *conflict*: a state and a lookahead at which more than one action is available. Conflicts come in two flavours, and they tell us different things about the grammar.

A *shift-reduce conflict* arises when a state contains both a complete item, ready to reduce on a lookahead a , and a partial item whose dot is just before a , ready to shift it. The classical example is the dangling else. The FRISCCc grammar contains both productions

$$\begin{aligned} \langle \text{if-statement} \rangle &\rightarrow \text{KR_IF } L_ZAGRADA \langle \text{expr} \rangle D_ZAGRADA \langle \text{stmt} \rangle \\ \langle \text{if-statement} \rangle &\rightarrow \text{KR_IF } L_ZAGRADA \langle \text{expr} \rangle D_ZAGRADA \langle \text{stmt} \rangle \text{KR_ELSE } \langle \text{stmt} \rangle \end{aligned}$$

so that an `if` may or may not be followed by an `else`. On the input `if (e1) if (e2) S1 else S2`, the parser, sitting just after reducing `S1` as the inner `if`'s body, has two equally legitimate moves. It could reduce the inner `if` right there, by the first production, leaving the `else` to attach to the outer `if`. Or it could shift the `else`, collecting evidence for the second production, and attach the `else` to the inner `if`. Both are valid LR moves. Both correspond to legitimate parses. The grammar is ambiguous at this point and no amount of lookahead resolves it.

The convention C adopts, and that almost every C-family language inherits, is *shift wins*: the `else` binds to the nearest `if`. FRISCCc implements this by a tie-breaking rule in `LRTTableBuilder.resolveConflicts`: when the table-builder encounters a shift–reduce conflict, it picks the shift and logs the resolution. The logging matters because shift–reduce conflicts have a habit of multiplying silently when the grammar grows; we want to know which states have been resolved by convention and which by intentional grammar design.

A *reduce–reduce conflict* is the deeper trouble. It arises when a state contains two complete items with the same lookahead, both ready to reduce by different productions. There is no principled way to choose: the two reductions would lead to different trees, with no clear precedence between them. Unlike shift–reduce, there is no convention that captures programmer intent. FRISCCc's table-builder does not refuse to produce a table; `LRTTableBuilder.resolveConflicts` breaks the tie by reducing with the production that appears earlier in the grammar definition (the one with the smaller grammar index) and logs a warning. The table is still produced and the parser still runs, but the resolution is essentially arbitrary — a warning that the grammar wants editing, not a meaning the grammar actually has.

Warning

A shift–reduce conflict resolved by “shift wins” is a deliberate ambiguity in the grammar that the parser papers over by convention. A reduce–reduce conflict is an accident the parser papers over by fiat. The two are reported separately by `LRTTableBuilder.java`: shift–reduce is tie-broken in favour of the shift, reduce–reduce in favour of the earlier production, and both log a warning while still producing a table. Treat shift–reduce resolutions as a small, contained debt; treat a logged reduce–reduce warning as a grammar bug to fix at once, because the production it silently dropped is one the grammar said it wanted.

There is a related question, halfway between conflicts and runtime errors, about what the parser does when the input is malformed. The action table marks impossible (state, terminal) combinations as errors; running into one is how the parser detects a syntax problem. The natural response is to stop and report the offending token, but that gives up after the first mistake, which is not helpful when the user is staring at a half-written program. A practical parser wants to recover — to discard a few tokens, jump to a state from which parsing can resume, and try to keep going.

The standard response to a syntax error is *panic-mode recovery*: on a syntax error the parser pops states until it finds one that can shift a *synchronisation token*, skips input until one of those tokens appears, and resumes parsing, so that one malformed statement does not abort the whole compile. FRISCcc already declares its synchronisation tokens — the third line of `config/parser_definition.txt` reads `%Syn TOCKAZAREZ D_VIT_ZAGRADA`, the semicolon and the closing brace, deliberately chosen because the semicolon ends a statement and the brace ends a block, so either is a sensible re-entry point. But the recovery itself is not yet implemented. `LRParser.handleError` reads the synchronisation tokens and then ignores them: it reports the offending token and throws a `ParseException`, so FRISCcc currently stops at the *first* syntax error and gives up. Wiring the synchronisation tokens into a real pop-and-skip loop — so the compiler can report several errors per run — is left as an exercise.

Conflicts and recovery together complete the engineering picture. The LR(1) construction either gives us a clean deterministic parser, in which case we are done, or it reports a list of conflicts, in which case we have a piece of grammar work to do. The shift–reduce conflicts get tie-broken by convention, the reduce–reduce conflicts get tie-broken with a warning that flags the grammar for editing, and a syntax error at runtime halts the parser at the first offending token (the synchronisation tokens needed for panic-mode recovery are declared but not yet acted on). Each mechanism is small, and each is exposed in the same configuration file so that a curious reader can examine all of them at once.

4.9 The FRISCcc grammar: precedence by stratification

It is time to look at the grammar we have been talking around. The specification file `parser_definition.txt` (in the `config/` directory) holds the entire syntax of FRISCcc’s C subset in 183 lines. The first three are header declarations — the set of non-terminals, the set of terminals, and the synchronisation tokens — and the remaining 180 are productions. The non-terminal names are in Croatian because the underlying course material is in Croatian; a glossary mapping them to their English equivalents lives in [Section A.2](#).

The original course materials this grammar derives from [27] use Croatian names for the grammar’s non-terminals — `<prijevodna_jedinica>` for translation unit, `<aditivni_izraz>` for additive expression, `<lista_specifikatora_kvalifikatora>` for the rather long list-of-type-specifiers-and-qualifiers, and so on. In this chapter we use the English names in prose; we cite the Croatian names only when we are quoting a literal line from the grammar file or naming a non-terminal you will see in the parse tree output. The full Croatian-to-English glossary, with one column per non-terminal and one column per terminal token, is in [Section A.2](#).

The grammar’s most striking feature is the precedence cascade. C has around fifteen levels of operator precedence; the FRISCCc subset keeps fifteen levels, one per non-terminal, and each level lives on its own non-terminal. From the loosest binding to the tightest, the cascade runs comma-expression, assignment, logical-or, logical-and, bitwise-or, bitwise-xor, bitwise-and, equality, relational, additive, multiplicative, cast, unary, postfix, primary. Fifteen non-terminals, one per precedence level, each connected to the next by a single pass-through production.

A pass-through production is the easy half. For each level X above the bottom of the cascade, the grammar has the production $\langle X \rangle \rightarrow \langle X_{\text{higher}} \rangle$. So `<aditivni_izraz>` has, among its alternatives, `<multiplikativni_izraz>` (the next level up in precedence), `<multiplikativni_izraz>` has `<cast_izraz>`, and so on all the way down to primary. A parse that crosses no operator at all walks straight down through every level of the cascade: when we parse the standalone identifier `a` as an expression, the parse tree contains `<izraz>` above `<izraz_pridruzivanja>` above `<log_ili_izraz>` above all the others, like a stack of nested boxes each containing exactly one child.

The other half of each level is the recursive case. For `<aditivni_izraz>`, in addition to the pass-through, the grammar has two productions for the addition and subtraction operators:

```

1  <aditivni_izraz>
2  <multiplikativni_izraz>
3  <aditivni_izraz> PLUS <multiplikativni_izraz>
4  <aditivni_izraz> MINUS <multiplikativni_izraz>

```

Two things are happening at once in these productions. The left-recursion (the same non-terminal appearing on the left of the right-hand side) encodes left-associativity: a sequence `a + b + c` parses with the leftmost plus deepest in the tree, because every `<aditivni_izraz>` on the left can recursively contain more additive expressions, but each `<multiplikativni_izraz>` on the right is closed — it cannot further left-recurse into addition. And the choice of `<multiplikativni_izraz>` on the right (rather than another `<aditivni_izraz>`) is what encodes precedence: the right operand of a plus can be a product, or a cast, or a unary, or any other higher-precedence form, but it cannot be another addition. The right side has to live at or below its parent’s precedence level.

Put both halves together and we get the full grammatical pattern for every level of the cascade. Each level has one pass-through production into the next-higher level, plus zero or more left-recursive productions for the operators that bind at this level. The pass-throughs let a primary expression rise to any level (by zero or more pass-through reductions); the recursive productions let operators of a level combine values of higher levels. The whole cascade is the way the grammar says “multiplication binds tighter than addition”, once, in writing, and the parser never has to know.

Figure 4.5 shows the descent of `a + b` through the cascade, exactly as the parser builds it for the anchor program of Chapter 3. The path is long — above the plus, the cascade rises through eight pass-through non-terminals, from `<izraz_pridruzivanja>` down to

<odnosni_izraz>, before reaching the additive level — but each of those is a single pass-through with one child, and the only node with two children is the <aditivni_izraz> where the plus actually lives.

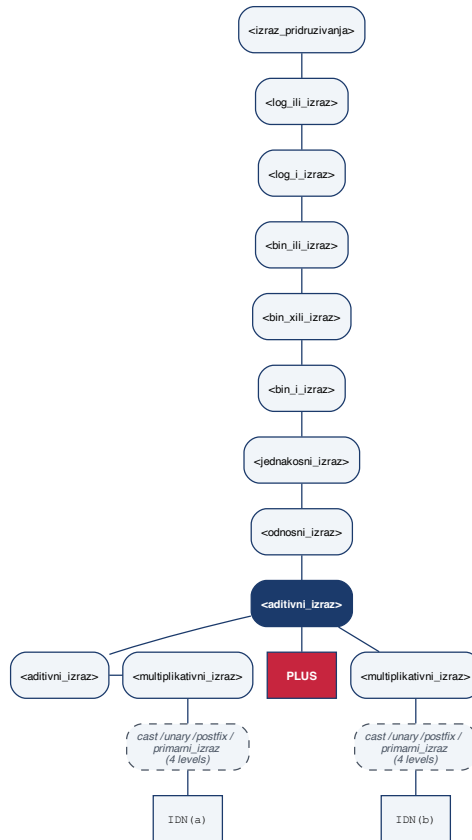


Figure 4.5: The precedence cascade for $a + b$ in the parse tree of `math_fibonacci_iter`. Above the plus, the cascade rises through eight pass-through non-terminals, from `<izraz_pridruzivanja>` down to `<odnosni_izraz>`; below it, the operands descend back to the primary leaves `a` and `b`. The shape is the grammar’s record of “where in the precedence hierarchy this expression sits”. The English names of every non-terminal are in [Section A.2](#).

The grammar’s other notable feature is the absence of typedef ambiguity. C’s grammar in the wild has the well-known declaration-versus-statement ambiguity: `a (b);` could be a function call `a(b)` treated as a statement, or it could be a declaration of `b` as a variable of type `a` (the parentheses around the declarator being redundant grouping), depending on whether `a` is a typedef name. The FRISCCc subset does not support typedefs, so this ambiguity

does not arise; every identifier is unambiguously a variable name, and the grammar is cleanly LR(1) without any lexer feedback. That simplification, combined with the explicit stratification, is what lets the canonical LR(1) construction produce a conflict-free table for the entire grammar in a single pass. That single pass is what the next section implements, class by class.

4.10 The implementation in FRISCcc

The Java code for the parser is split across three packages, and the split mirrors the structure of this chapter. The `grammar` package reads `config/parser_definition.txt` into memory, organises it into productions, and computes the FIRST sets we need for closure. The `lr` package builds the canonical collection of LR(1) item sets, fills in the ACTION and GOTO tables, and runs the parsing loop. The `table` package is the on-disk representation of the tables, with a cache layer that avoids the cost of rebuilding them on every compiler invocation.

The reading is unremarkable. `Grammar.java` holds the in-memory representation: a list of productions, indexed both by left-hand side and by integer. `GrammarParser.java` reads the specification file, expands the header sections into terminal and non-terminal sets, and validates that every symbol in every production is declared in the header. `FirstSetComputer.java` runs the standard FIRST-set fixpoint: initialise the set for each terminal to itself, the set for each non-terminal to empty, iterate, propagate. The iteration is a single while loop that terminates when no FIRST set changes for a full pass — the monotonicity of the update guarantees we reach the fixpoint in finitely many iterations.

The building is where the chapter's theory lives. `LRItem.java` is the record for a single item: a production, a dot position, a lookahead set. `LRItemSet.java` is a set of items with the small piece of equality-and-hashing logic we need to compare item sets for membership in the canonical collection. `LRClosure.java` and `LRGoto.java` implement the two operations of [Section 4.5](#), each as a short re-scan-until-stable fixpoint loop (equivalent to the worklist form the chapter draws). `LRTableBuilder.java` runs the canonical-collection construction — start from the augmented start item, repeatedly take goto on every grammar symbol, add new item sets to the worklist, continue until convergence — and then walks the collection once to fill in the ACTION and GOTO tables, applying the shift-wins tie-break on shift-reduce conflicts and the earlier-production tie-break on reduce-reduce conflicts, logging a warning for each.

The runtime is small. `LRParser.java` holds the state stack, the lookahead, and the partial parse tree, and runs the loop we saw in [Section 4.7](#): consult the table, shift or reduce, update the tree. `Parser.java` is the entry point that wires it all together: read the grammar, ensure the table is built (or cached), open the token stream, run the parser, write the parse tree to `compiler-bin/ast.txt`. The class is fewer than two hundred lines.

The cache is worth a paragraph. Building the canonical collection for the FRISCcc grammar takes on the order of several seconds: a thousand-odd item sets, each with a few dozen

items, each closure running its own worklist, and the whole thing iterating until convergence. We do not want to pay that cost on every invocation of the compiler. So `LRTableCache.java` serialises the built table to `target/parser-cache/lr_table.ser` and, on subsequent runs, reloads it whenever that file exists. The cache is keyed purely on existence: it does not hash the grammar or compare anything against `parser_definition.txt`. An unchanged grammar loads the table in milliseconds, and the construction is amortised across the project's lifetime, paid for once at the cost of a few seconds. The flip side of that simplicity is that the cache cannot notice a grammar edit; after changing the grammar we must clear it by hand — delete `target/parser-cache/lr_table.ser` (or call `clearCache`) — or the parser will keep loading the stale table.

Engineering tip

If the parser starts behaving oddly after a grammar edit, the first thing to check is whether you are running against a stale cache. Because the cache only checks that `lr_table.ser` exists — it never re-reads the grammar — a changed `parser_definition.txt` will keep loading the old table until you clear it. Delete `target/parser-cache/` (or call `clearCache`) to force a clean rebuild, which is also the cure if the cache file is corrupted (rare, but possible if a build is interrupted). The cache is strictly a performance optimisation; it never holds the authoritative parser state.

What is striking about reading the implementation alongside the chapter is how directly each piece of theory corresponds to a piece of code. The FIRST-set computation is one class; closure is one class; goto is one class; the table-builder is one class with two methods; the parser driver is one class with one main loop. The hard thinking is in the theory; the code is the theory written in Java, with a small amount of caching machinery wrapped around the perimeter. The next section puts the whole stack to work: we give `LRParser.java` the token stream from [Chapter 3](#) and watch it build the parse tree for `math_fibonacci_iter`.

4.11 Worked example: parsing the iterative Fibonacci

We turn now to the program we have been promising to parse. The source lives under `examples/real_world/` in the `math_fibonacci_iter` directory, and the token stream sits in `compiler-bin/tokens.txt` from the last chapter; the parser takes the latter as input and writes its output to `compiler-bin/ast.txt`. The full tree is several hundred nodes, far too many to draw on one page, so we will look at it in two pieces: the top-level skeleton in [Figure 4.6](#), and an excerpt of the textual dump in [Listing 4.1](#).

The skeleton is exactly what the grammar dictates. The start symbol `<prijevodna_jedinica>` (translation unit) sits at the root. Its single child is an `<vanjska_deklaracija>` (external declaration), which in turn has a single child, a `<definicija_funkcije>` (function definition). The function definition expands into a return type, a declarator (the name `main` and its parameter list), and a `<slozena_naredba>` (compound statement). The compound statement is where the function body lives, and it expands into a list of local declarations followed by

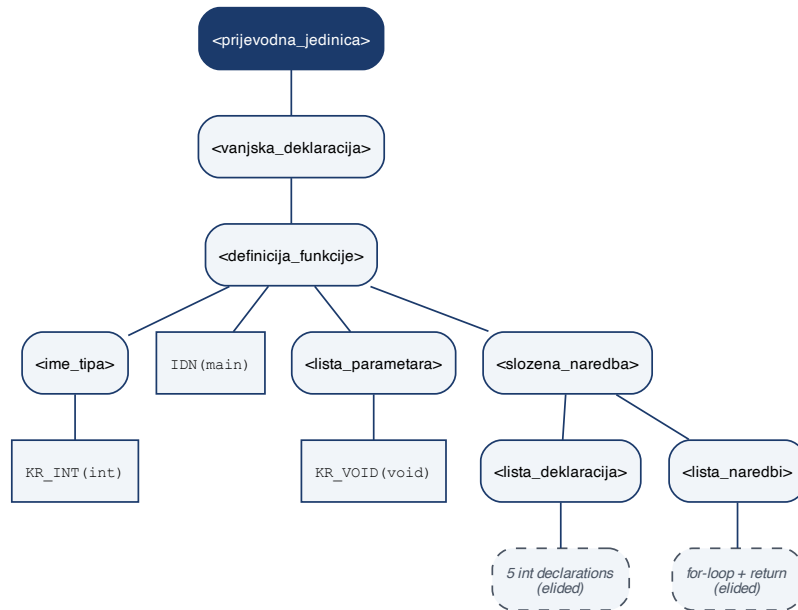


Figure 4.6: Top-level skeleton of the parse tree the parser builds for `math_fibonacci_iter`. The translation unit at the root contains one external declaration, which is the function definition for `main`. The function definition has a return type (`int`), a name (`main`), a parameter list (the single token `void`), and a compound statement holding the body. The compound statement, in turn, contains a list of local declarations (the five `int` declarations at the top of `main`) and a list of statements (the `for` loop and the `return`). The figure shows only the top three levels of the tree; the expansion of the compound statement's children is given in [Listing 4.1](#).

a list of statements. Every interior node corresponds to a production from the grammar; every leaf is a token from the stream we left at the end of [Chapter 3](#).

[Listing 4.1](#) reproduces the first forty-seven lines of the parser’s textual dump of the tree. Each line is one node; the indentation reflects the node’s depth in the tree. Non-terminal labels are in angle brackets; terminals are printed as `KIND , lexeme`, where the kind is the lexer’s token class and the lexeme is the original source text. The textual dump omits the source line for compactness, but each terminal leaf in the in-memory parse tree still carries the line number the lexer attached to it; it is the parser’s gift to every later phase. When the semantic analyser reports a type error, the diagnostic will quote the source line at which the offending identifier was scanned, courtesy of the leaf node at that point in the tree.

Two things in this tree are worth pausing on, because they are the points the rest of the book will keep returning to.

The first is the precedence cascade we have just been discussing. Look at where the parser puts the `= 20` that initialises the local variable `n`. The initialiser is a `<inicijalizator>`, which in turn is an `<izraz_pridruzivanja>` (assignment expression), which is a `<log_ili_izraz>` (logical-or expression), which is a `<log_i_izraz>` (logical-and expression), and so on, all the way down through fourteen levels of pass-through until we hit the primary expression `BROJ , 20` at the bottom. Fourteen nodes, each with one child, encoding “this is an expression sitting at the assignment level of precedence, but containing no operator above primary”. The chain looks heavy, but it is exactly the information every later phase will want: the precedence level at which each sub-expression sits, recorded directly in the shape of the tree. The interior `+ in t = a + b` sits inside some chain of pass-throughs at exactly the additive level, and nowhere else; the parser cannot have misplaced it, because the grammar would not allow it. Every operator in the source ends up at exactly the right level of the tree, by construction.

The second is the persistence of source location through every terminal. Every leaf in [Listing 4.1](#) carries a source line. The lexer set it; the parser preserved it; the semantic analyser will read it. When something goes wrong in `math_fibonacci_iter` — a type mismatch, an undeclared variable, an unreachable statement — the diagnostic will quote a line number, and the line number will come from one of these leaves. The contract that the lexer offered the parser is the contract the parser passes on to everything that follows. The tree is the shared state of the front end; the line numbers and lexemes are the breadcrumbs that lead back to the source.

Key insight

The parse tree is the front end’s contract with the rest of the compiler. Its shape encodes the grammar’s precedence and associativity decisions; its leaves preserve the lexer’s source locations and lexemes. Every later phase walks the tree without needing to look at the original tokens again, and every diagnostic the compiler produces traces back to a leaf via the line number embedded in it. Get the tree right, and the rest of the compiler inherits its correctness for free.


```

1  <prijevodna_jedinica>
2    <vanjska_deklaracija>
3      <definicija_funkcije>
4        <specifikatori_deklaracije>
5          KR_INT , int
6        <deklarator>
7          <izravni_deklarator>
8            <izravni_deklarator>
9              IDN , main
10             L_ZAGRADA , (
11             <lista_parametara>
12               <deklaracija_parametra>
13                 <specifikatori_deklaracije>
14                   KR_VOID , void
15                 D_ZAGRADA , )
16             <slozena_naredba>
17               L_VIT_ZAGRADA , {
18               <lista_deklaracija>
19                 <lista_deklaracija>
20                   <lista_deklaracija>
21                     <lista_deklaracija>
22                       <lista_deklaracija>
23                         <deklaracija>
24                           <specifikatori_deklaracije>
25                             KR_INT , int
26                           <lista_init_deklaratora>
27                             <init_deklarator>
28                               <deklarator>
29                                 <izravni_deklarator>
30                                   IDN , n
31                                   OP_PRIDRUZI , =
32                                 <inicijalizator>
33                                   <izraz_pridruzivanja>
34                                     <log_ili_izraz>
35                                       <log_i_izraz>
36                                         <bin_ili_izraz>
37                                           <bin_xili_izraz>
38                                             <bin_i_izraz>
39                                               <jednakosni_izraz>
40                                                 <odnosni_izraz>
41                                                   <aditivni_izraz>
42                                                     <multiplikativni_izraz>
43                                                       <cast_izraz>
44                                                         <unarni_izraz>
45                                                           <postfiks_izraz>
46                                                             <primarni_izraz>
47                                                               BROJ , 20

```

Listing 4.1: The first forty-seven lines of the parser’s textual dump of the parse tree for `math_fibonacci_iter`, as written to `compiler-bin/ast.txt`. Indentation reflects tree depth; terminals appear as `KIND , lexeme`. Source line numbers are tracked on the in-memory nodes but elided here for compactness. The excerpt covers the function header and the descent of the initialiser `= 20` through the precedence cascade down to its leaf `BROJ , 20`; the full dump is several hundred lines. The Croatian non-terminal names are translated in [Section A.2](#).

We have a tree. The grammar’s stratification has put every operator at the right level; the LR(1) construction has produced a parser that builds the tree without ambiguity; the implementation has cached the tables so the cost is amortised; the leaves carry their source locations. The next chapter starts where this one ends. With a tree in hand and contracts honoured, the question becomes: is the program the tree represents *well-typed*? The semantic analyser of [Chapter 5](#) walks the same tree, attaches type information to every interior node, and checks the rules of the language. The parser said “this is a syntactically valid program”; the semantic analyser will say whether the program makes sense.

Your turn

The companion repository’s next module is `ch04-parser`, and it picks up where the `ch03-lexer` module left off: with a working tinyC lexer and a token stream ready to be parsed. The tinyC grammar is a simplified version of FRISCCc’s — integer types only, no structs, no pointers, no casts — but it preserves the precedence cascade and the if-else dangle, so the lab teaches the same lessons the chapter does on a smaller surface. From the companion root, `cd ch04-parser` and read `README.md` for the lab’s specifics. The starter at `ch04-parser/starter/src/main/java/com/frisccc/tinyc/parser/` contains stubbed-out `Closure.java`, `Goto.java`, and `LRTTableBuilder.java` classes; your job is to fill in the worklist loops for closure and goto, build the canonical collection for the tinyC grammar, and feed the resulting ACTION/GOTO tables to the parser driver that is already provided. The tests in `ch04-parser/tests/` check three things in increasing order of difficulty: that closure on a single seed item produces the expected item set, that goto on a few representative (state, symbol) pairs produces the expected successor states, and that the full parser, driven by your tables, parses the tinyC version of `math_fibonacci_iter` into the same tree the solution produces. Run `mvn test -pl ch04-parser` from the companion root; when all three test classes pass, come back here for the chapter’s practice problems.

Practice

Exercise 4.1 (Applied · ★☆☆)

The three-rule expression grammar of [Section 4.1](#) has three non-terminals: E , T , and F . Compute $\text{First}(E)$, $\text{First}(T)$, and $\text{First}(F)$ by iterating the monotone fixpoint update: start with every set empty, apply the FIRST rules once, record what changed, and iterate until nothing changes. Show the table after each round, and say how many rounds pass before the sets stop changing. Explain in one sentence why a grammar whose non-terminals depend on one another in a chain — here E on T on F — needs more than a single round to stabilise.

Exercise 4.2 (Applied · ★★★)

Take the initial item set I_0 of the three-rule expression grammar, computed in [Section 4.5](#). Compute $\text{goto}(I_0, T)$ by hand: list every item in I_0 whose dot sits immediately before the non-terminal T , advance each dot one position, and take the closure of the resulting kernel. Write the final item set as a list of LR(1) items of the form $[A \rightarrow \alpha \cdot \beta, a]$. Compare your result to the description in [Section 4.5](#) and explain any difference in lookahead sets.

Exercise 4.3 (Applied · ★★★)

Trace the LR(1) parser on the input $(a + b) * c$ using the ACTION/GOTO table of [Table 4.1](#). Show every step as a triple (stack, remaining input, action taken), the same format as the trace in [Section 4.7](#). Once the trace is complete, draw the resulting parse tree and compare it to the tree for $a + b * c$: identify the one node in the tree whose subtree structure differs, and explain in two sentences why the parentheses force that difference.

Exercise 4.4 (Applied · ★★★)

Take the dangling-else fragment `if (e1) if (e2) S1 else S2`. Draw both parse trees the grammar admits: the one where `else` binds to the inner `if` and the one where it binds to the outer. Then identify the parser state in which FRISCc's table-builder reports a shift–reduce conflict on lookahead `KR_ELSE`, and explain in two sentences why the “shift wins” tie-break selects the inner-if tree rather than the outer-if tree.

Exercise 4.5 (Applied · ★★★)

Suppose we rewrote the additive expression production as

$$\langle \text{add} \rangle \rightarrow \langle \text{mul} \rangle \mid \langle \text{mul} \rangle + \langle \text{add} \rangle$$

putting the recursion on the *right* instead of the left. Argue, without constructing the full canonical collection, why the resulting grammar still generates every additive expression — and yet produces parse trees with the *opposite* associativity to the left-recursive version. Which concrete tree does $a - b - c$ produce under right recursion? Is that the meaning a C programmer expects, and what would the numeric result be for $a = 4$, $b = 3$, $c = 1$?

(This is the chapter's stretch exercise; the argument about associativity can be made precise in about half a page.)

Project

Exercise 4.6 (Lab · ★★★)

Extend the tinyC parser in `frisccc-companion/ch04-parser/` to support the comma operator `,` as a low-precedence binary operator that evaluates its left operand for side effects and returns its right. The tinyC grammar currently has no comma expression; the change introduces a new non-terminal at the very top of the precedence cascade, above `<assignment_expr>`, and adds a pass-through production so that every existing expression still parses.

Open `ch04-parser/starter/src/main/java/com/frisccc/tinyc/parser/` and locate `Closure.java`, `Goto.java`, and `LRTTableBuilder.java`. Before adding the comma extension, make sure the three core worklist loops are implemented and all baseline tests pass (`mvn test -pl ch04-parser`). Then, in the grammar resource file `ch04-parser/starter/src/main/resources/tinyc_grammar.txt`, add the two productions

```
<comma_expr> ::= <assignment_expr>
<comma_expr> ::= <comma_expr> , <assignment_expr>
```

and update the start symbol so that `<comma_expr>` is the new top of the expression hierarchy. Rebuild the ACTION/GOTO tables by re-running `LRTTableBuilder` and check whether any new shift-reduce or reduce-reduce conflicts appear in its report. Run `mvn test -pl ch04-parser -Dtest=CommaExprTest`; the test feeds the program fragment (`a = 1, b = 2`) to your parser and expects the resulting tree to have the comma operator at the root with two assignment-expression children beneath it. Then run the full suite (`mvn test -pl ch04-parser`) to confirm no regressions.

Once the tests pass, answer three questions in writing. First, did your change introduce any new conflicts, and if so, how did you resolve them? Second, standard C gives the comma operator even lower precedence than assignment; did your grammar end up matching that convention? Third, what does the parse tree of `a, b = 1` look like under your grammar, and is the result left- or right-associative?

Stretch: add support for the comma operator in function argument lists (where it separates arguments, not evaluates left-for-side-effects), and explain in writing how you distinguish the two uses in the grammar.

Notes and further reading

The canonical LR parsing technique is due to Donald Knuth [48], whose 1965 paper proves both that $LR(k)$ is the largest class of context-free grammars admitting a deterministic table-driven parser and that the canonical construction is sound and complete for it. The paper is short and worth reading even if your only use for it is as a piece of intellectual history; the construction in section 4 is essentially the one we built piece by piece. The pragmatic objection to the canonical construction is its state count, which grows quickly with grammar size, and the standard responses — $SLR(1)$ by DeRemer [24] and $LALR(1)$ by DeRemer and Pennello [23] — merge canonical states whose core $LR(0)$ items agree, at the cost of slightly weaker lookahead discrimination. FRISCCc uses the full canonical

construction for pedagogical clarity; the tables are larger than LALR(1) would produce, but the construction is uniform and the per-state logic is the one we just walked through.

The textbook references for the construction are the Dragon Book [2] — Aho, Lam, Sethi, and Ullman’s *Compilers: Principles, Techniques, and Tools* — and Grune and Jacobs’s *Parsing Techniques: A Practical Guide* [33]. Chapter 4 of the Dragon Book is the standard treatment of LR parsing in the literature, with full constructions for LR(0), SLR(1), LALR(1), and LR(1), and a careful proof of the determinism theorem. Grune and Jacobs is more recent and noticeably more opinionated; it covers LR parsing alongside every other parsing technique and is the closest thing to an encyclopedia the field has. Either book is a fine companion to this chapter, and between them they have a proof of every theorem we have stated and most of the ones we have not.

For the practical history, Stephen Johnson’s *Yacc* [41] is the parser generator that turned LALR(1) into an industry standard. Yacc’s input format inspired Bison, ANTLR’s grammar grammar, FRISCCc’s `parser_definition.txt`, and essentially every academic parser generator of the last fifty years. Reading the original Yacc technical report is the cheapest way to see why our specification file looks the way it does — the structure, the header sections, the rule-per-line layout, the percent-prefixed declarations are all inherited from Yacc.

The Earley algorithm [25] and Tomita’s generalised LR (GLR) parser [81] are the standard fallbacks when the LR(1) construction reports conflicts that cannot be cleanly resolved. Earley accepts every context-free grammar in cubic time and is the standard choice for languages with genuinely ambiguous syntax (natural-language fragments, mathematical-notation parsers, some research languages). GLR extends LR(1) by maintaining multiple parse stacks in parallel when conflicts arise, accepting any input in cubic time and most inputs in linear time. FRISCCc does not use either, because the FRISCCc grammar is clean LR(1) without exception. But if a language designer ever needs to push past LR(1), Earley and GLR are where to look.

Finally, the original PLT-FER course materials that supplied FRISCCc’s grammar specification [27] are the Croatian-language reference for the non-terminal names and the specification-file format itself. Section A.2 translates every name we used in this chapter to its English equivalent. For the underlying theory of context-free languages, Hopcroft, Motwani, and Ullman’s *Introduction to Automata Theory, Languages, and Computation* [39] and Sipser’s *Introduction to the Theory of Computation* [75] are the standard references; both contain the Chomsky hierarchy, the pumping lemma for context-free languages, and the closure properties we mentioned in passing.

Chapter 5

Meaning, not just shape: semantics

“A type system is the simplest formal method that actually works on real programs.”
— Benjamin C. Pierce, *Types and Programming Languages*, 2002

Open a fresh C file and type `5 = x;`. Then add a second line: `return y;`, where nothing called `y` was ever declared. Feed the result through the parser of [Chapter 4](#) and the parser will not complain. The first line has the shape of an assignment — expression, equals sign, expression, semicolon — and the C grammar permits exactly that shape. The second line has the shape of a return statement, and the grammar is happy with any identifier in the expression slot. Both programs are syntactically well-formed; both are also nonsense. The first assigns to a literal, which is not a place where anything can be stored. The second names a variable that does not exist.

The parser cannot see either error, because both are statements *about meaning* — about what the identifiers refer to, what kinds of values the expressions produce, where assignments are allowed — and the parser’s job is exclusively the shape. The contracts the grammar cannot express are what we call *static semantics*, and enforcing them is the job of this chapter’s pass.

We will spend the chapter learning what those contracts look like, how we write them down formally as typing rules, and how FRISCCc enforces them by walking the parse tree post-order and decorating each node with the type and category attributes the rest of the compiler relies on. The anchor is the same iterative Fibonacci we tokenised in [Chapter 3](#) and parsed in [Chapter 4](#); this time we will watch the analyser push a scope when the for loop opens, look up `a` and `b` as integers, type `a + b` as `int`, and pop the scope when the loop closes. By the end of the chapter the program will be *well-typed*, and the artifact that flows downstream to [Chapter 6](#) will be the parse tree decorated with everything the IR generator needs to know.

5.1 What the parser cannot know

Let us look at three small programs that the C grammar accepts but that violate a contract living outside the grammar. The analyser rejects the first two; the third trips a contract

FRISCCc does not yet enforce, and we will use it to mark the boundary of what the analyser checks.

The first is the assignment to a literal we just saw:

```
1  int main(void) {
2      5 = x;
3      return 0;
4  }
```

The grammar says an *assignment expression* is an expression on the left, an equals sign, and an expression on the right. Both 5 and x are valid expressions; the rule fires. But 5 is not a place; it is a *value*. You cannot put something into the literal five any more than you can put something into the number sixty-two. The distinction is between expressions that denote a location — *lvalues*, called that because they may appear on the left of an assignment — and expressions that denote only a value — *rvalues*. The grammar treats both the same. The semantic analyser does not.

The second is the use of an undeclared identifier:

```
1  int main(void) {
2      return y;
3  }
```

y is a syntactically valid identifier. The grammar’s production for the return statement asks for an expression after the keyword; an identifier is an expression; the rule fires. The grammar has no way of asking “is this identifier declared anywhere visible?”, because that question concerns the *environment* the identifier appears in, not the shape of the production. Names exist in *scopes*, and the analyser must keep track of which names are visible where.

The third is more subtle:

```
1  int main(void) {
2      int x = 5;
3      while (x) {
4          if (x == 0) break;
5          return x;
6      }
7  }
```

The program parses without complaint. The grammar guarantees nothing about whether break appears inside an enclosing loop — it does here, so this particular break is fine — and nothing about whether every path through main reaches a return. The path where the loop body finishes without break or return falls off the end of main, which a function that promises to return int ought not to do. These are *flow-sensitive* invariants: they

depend on the structure of the statements rather than the shape of any one expression, and they require an analyser to track context that the grammar throws away as soon as a rule reduces. FRISCCc’s analyser checks the first of them — `break` and `continue` must sit inside a loop — but, as we will see in [Section 5.5](#), it does *not* enforce the returns-on-every-path contract; that one we leave as an exercise to add.

Three contract violations, three flavours of contract. The first is about *categories*: which expressions are places. The second is about *environments*: which names mean what, where. The third is about *statement context*: which constructs are allowed in which positions in the control structure. None of them are in the grammar’s reach. The first two are squarely in the analyser’s; the third sits at its edge. The clean way to handle the first two together is to give every expression a *type* — not just `int` or `char` but a richer attribute that records both the kind of value and the category — and to enforce the rules in terms of those types.

5.2 Types as the shape of meaning

Look at the third line of the anchor program, `int n = 20;`. To the parser this was a declaration: a type specifier, an identifier, an initialiser. To the semantic analyser it is something larger. The specifier `int` stamps a type onto the name `n`; the value `20` comes with its own type, namely the type of integer literals; the declaration is well-formed only if those two types fit together. “Fit together” will turn out to mean something precise. First we need a vocabulary of types.

The C subset compiled by FRISCCc has a small, fixed set of types, and every expression in any program will end up labelled with one of them. The primitives are integers (`int32`), single characters (`char`), Q16.16 fixed-point floats (`float`) which we will meet properly in [Chapter 9](#), and the empty result type (`void`). The compound forms are pointers (`ptr< $\tau$ >`), arrays (`array< $\tau$ ,  $n$ >`), and named structs (`struct N`). The type set is closed under those constructors — a pointer to an array of structs is a perfectly good type — but small enough that we can list all of them on one page and keep them in our heads.

Formally, the set $\mathcal{T}$ of FRISCCc types is the smallest set containing `int32`, `char`, `float`, and `void`, and closed under three constructors: `ptr< $\tau$ >` $\in \mathcal{T}$ for any $\tau \neq \text{void}$; `array< $\tau$ ,  $n$ >` $\in \mathcal{T}$ for any $\tau \neq \text{void}$ and any positive integer n ; and `struct N` $\in \mathcal{T}$ for any struct name N declared in the source. There is no boolean type at this layer — the source never spells the word “bool”, and the analyser follows C in typing comparison and relational results as `int32` (0 for false, 1 for true). The typed IR of [Chapter 6](#) does introduce a dedicated bool type, lifting those ints into it so it can thread the result explicitly through every conditional branch. The type `float` denotes the Q16.16 fixed-point representation we will define properly in [Chapter 9](#); for now it is enough to know that it is one of the numeric types and that mixing it with the others requires a conversion.

With the type set in hand we can start labelling. Walk through the anchor program with the analyser:

```
3  int main(void) {  
4      int n = 20;  
5      int a = 0;  
6      int b = 1;  
7      int i;  
8      int t;  
9  
10     for (i = 0; i < n; i++) {  
11         t = a + b;  
12         a = b;  
13         b = t;  
14     }  
15  
16     return a;  
17 }
```

The header `int main(void)` declares `main` as a function with no parameters returning `int`. The analyser records a function-type entry `void → int32` in the global scope. Inside the body, each declaration — `int n = 20;`, `int a = 0;`, `int b = 1;`, `int i;`, `int t;` — gives the analyser a name and a type to record. The integer literals `0`, `1`, `20` all type to `int32`. Each initialiser type-checks because `int32` “fits” `int32`; we will say what fits means below. The bare `int i;` and `int t;` declarations introduce names without initialisers, which is fine for local variables — the analyser does not require initialisation, and [Chapter 6](#) accordingly generates code that leaves the slots uninitialised.

The loop header is where types start doing real work. The guard `i < n` has both operands of type `int32`; the comparison rule gives the result type `int32` (0 or 1), which [Chapter 6](#) will lift into a `bool` when it lowers the conditional branch. The update `i++` is the post-increment of an `int32`-typed lvalue, which is again `int32`. The body’s `t = a + b;` is the binary-arithmetic rule on `int32`-typed operands giving an `int32`-typed result, then the assignment rule storing an `int32` into an `int32`-typed lvalue `t`. Everything fits. Finally `return a;` ships an `int32`-typed value back to a function whose declared return type is `int32`. Every expression in the program ends up wearing a type, every contract is satisfied, and the program is well-typed.

The word “fits” has been carrying a lot of weight. The promotion relation $\tau' \prec \tau$ makes it precise. We allow `char` to promote implicitly to `int32` (sign-extension on the FRISC machine word), and we allow `int32` to promote implicitly to `float` (lifted into `Q16.16` by the runtime helper). We allow each type to “promote” to itself, trivially. We do not allow the reverse directions implicitly: shrinking `int32` to `char` or `float` to `int32` requires an explicit cast, and pointer conversions require explicit casts too. This makes FRISCcc’s subset stricter than ISO C; we take the trade because the losses we forbid are exactly the ones a reader of the source would have to think twice about anyway.

Key insight

A type is a static prediction about a runtime value. The analyser’s contract with the IR generator is that the prediction holds: an expression labelled `int32` produces a 32-bit signed integer at run time — including a comparison result, which is just the `int32` values 0 or 1. [Chapter 6](#) relies on this contract every time it picks an opcode. The whole reason this chapter does anything at all is so the back end never has to look at a node and wonder.

Types are half the analyser’s vocabulary. The other half is the collection of bindings the analyser has to remember as it descends the tree — the data structure that answers “does y mean anything here?”. That is the symbol table, and it is where we go next.

5.3 Symbol tables and scope

When the analyser arrives at the identifier `a` inside the loop body’s `a + b`, it needs to answer two questions: is `a` declared anywhere visible right now, and if so what is its type? A single hash table mapping name to type would handle the first half but miss the point of *scope*: the same identifier `a` can be a local in one function, a different local in another, a parameter in a third, and an outer variable shadowed by a loop counter in a fourth. The analyser cannot answer with a flat map; it has to remember which declarations are currently visible and which are temporarily covered.

The classical solution is a *scope stack*, also called a *lexical environment*. Picture a tower of dictionaries. At the bottom sits the global scope, holding everything declared at file level — in our case, just `main`. On top of it sits the function scope for whatever function is currently being analysed, holding its parameters. On top of that sits the scope of the function’s body block; on top of that the scope of any nested block; and so on. When the analyser enters a new block it pushes a fresh dictionary onto the tower; when it leaves the block it pops the top dictionary off. Looking up a name walks the tower from the top down and returns the first match, which is exactly the semantics of lexical scoping in C.

[Figure 5.1](#) shows the tower at the moment the analyser is staring at the assignment in the loop body. Notice that the for-body block is empty in this program because nothing is declared inside the braces; only the loop’s update variable and the locals from the surrounding function body are visible. If we had written `for (int i = 0; ...)` — C99 syntax FRISCCc does not accept — there would have been an additional intermediate scope for the loop header itself, but FRISCCc’s subset follows the older discipline of declaring all locals at the top of the enclosing block.

The mechanics of the stack are simple enough to state precisely.

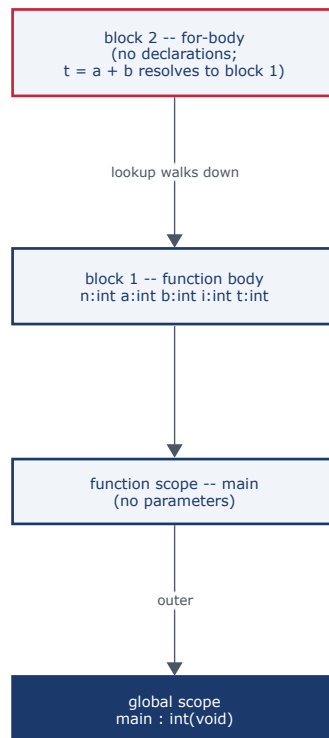


Figure 5.1: The scope stack at the moment the analyser is processing `t = a + b`; inside the `for` loop body of `math_fibonacci_iter`. The global scope at the bottom holds the single binding for `main`; the function scope is empty because `main` has no parameters; block 1 (function body) holds the five locals `n`, `a`, `b`, `i`, `t`; block 2 (the `for`-body) sits on top of it. Lookup walks the tower from top down and finds the first match.

A *typing context* Γ is modelled as a non-empty list of *frames*:

$$\Gamma = \Gamma_{\text{global}} ; \Gamma_{\text{function}} ; \Gamma_{\text{block}_1} ; \dots ; \Gamma_{\text{block}_k}$$

Each frame is a finite map from identifiers to symbols (variables get a type and a const flag; functions get a signature). The empty context contains only Γ_{global} . The operations are: *push* a fresh empty frame on entering a block; *pop* the top frame on leaving a block; *insert* a binding $x : \tau$ into the top frame, failing if x already exists in the top frame (redeclaration in the same scope is an error); and *lookup* of x , which walks frames from top to bottom and returns the first binding found, failing if none does. The *typing judgment* $\Gamma \vdash e : \tau$ asserts that, under context Γ , the expression e has type τ . The rules of Section 5.4 are stated in terms of this judgment.

Stated as three small algorithms, the mechanics fit on half a page.

Algorithm 5.1: Lexical-scope stack operations: EnterScope, LeaveScope, Insert, and Lookup. Push and pop track block nesting; insert respects in-scope redeclaration; lookup walks the stack top-down and returns the first match.

Input: a stack Γ of scope frames; an identifier x ; a type τ (for the binding case)

Output: side-effects on Γ , or a symbol (for the lookup case)

```

1 procedure EnterScope( $\Gamma$ );
2   push an empty frame onto  $\Gamma$ ;
3 procedure LeaveScope( $\Gamma$ );
4   pop the topmost frame from  $\Gamma$ ;
5 procedure Insert( $\Gamma, x, \tau$ );
6   let  $F$  be the top frame of  $\Gamma$ ;
7   if  $x \in F$  then
8     | report “redeclaration of  $x$  in current scope”;
9   bind  $F[x] \leftarrow \tau$ ;
10 function Lookup( $\Gamma, x$ );
11   for each frame  $F$  in  $\Gamma$  from top to bottom do
12     | if  $x \in F$  then
13       | | return  $F[x]$ ;
14   report “undeclared identifier  $x$ ”;

```

Algorithm 5.1 states the four operations the analyser performs on the stack. FRISCCc’s analyser implements them in the `SymbolTable` class under `compiler-semantics/.../symbols/SymbolTable.java`. The class models the tower as a tree of scopes rather than a literal stack: each scope holds a reference to its parent and a list of its children, and the analyser keeps a moving “current scope” pointer that descends and ascends as it walks the parse tree. The tree representation gives the implementation two practical benefits. It

makes scope debugging easy — the children are recorded permanently rather than thrown away on pop, so the analyser can dump the whole tree at end-of-run — and it lets the IR generator of [Chapter 6](#) walk the same scope structure when it lays out stack frames.

The discipline that makes the stack work is matched push/pop. Every time the analyser visits a node that opens a new block — a function body, a compound statement after a control structure, a nested block within a function — it descends into a child scope, processes the block's contents, then ascends back to the parent. Mismatched pushes and pops would corrupt the visible-name set for everything downstream. The visitor pattern of [Section 5.6](#) encodes the discipline structurally: every block-visiting method calls `enterChildScope` on the way in and restores the previous scope on the way out, using Java's `try/finally` so that pop happens even when an exception interrupts the visit.

Lookup, by contrast, is read-only: the analyser walks down the parent chain looking for the first binding of the given name, and either finds one (returns its symbol) or runs off the bottom (reports an undeclared-identifier error). The walk is bounded by the depth of nesting, which in practice is two or three frames for the kind of programs in our example corpus and not more than ten for any program that fits on a page. The cost of lookup is therefore $O(\text{nesting})$ times the cost of one hash-table probe, which is bounded by a small constant in real programs.

Engineering tip

The trickiest bug in a hand-written semantic analyser is processing a block's declarations *outside* the block's own scope. If the analyser pushes the new scope only after processing declarations, the declarations land in the parent and the block's own statements cannot see them. FRISCCc avoids this by pushing the scope on entry to the compound statement node and processing every child — declarations and statements alike — under the new scope. The reference in the parser grammar lives at `<slozena_naredba> ::= L_VIT_ZAGRADA ...D_VIT_ZAGRADA`, and the visitor method for that production opens its scope before walking the brace's contents.

With names that can be looked up and types they can be looked up to, we have everything we need to state the rules of the analyser. The rules are infinitely many in principle — one for every well-formed expression — but generated by a small, structurally-inductive recipe: a typing calculus.

5.4 Typing rules as a calculus

Watch the analyser type the expression `a + b` inside the loop body. `a` is an identifier; the lookup rule finds it in block 1 with type `int32`. `b` is an identifier; the lookup rule finds it in block 1 with type `int32`. The plus sign is the binary-arithmetic operator, and the rule for binary arithmetic on two `int32`-typed operands gives an `int32`-typed result. Three rule applications, arranged into a small tree:

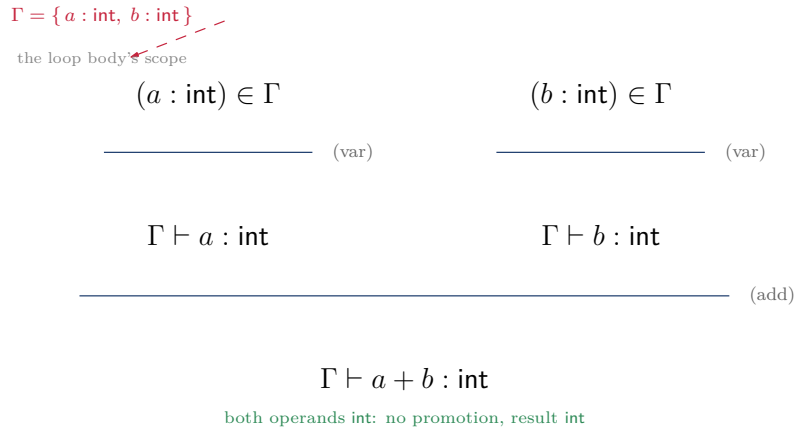


Figure 5.2: The typing derivation for the expression `a + b` inside the `for` loop body of the anchor program. The context Γ is shown with its block-1 bindings ($a : \text{int32}$, $b : \text{int32}$, and the rest of the loop’s locals), and the two leaves are the `(var)` look-ups that read a and b out of it: $\Gamma \vdash a : \text{int32}$ because $a : \text{int32}$ is in the block-1 frame of Γ , and similarly for b . Those two look-ups feed the single `(add)` inference step at the root, which applies the binary-arithmetic rule for `int32`-typed operands, concluding $\Gamma \vdash a + b : \text{int32}$. The bar above each conclusion separates premises (above the bar) from conclusion (below). Every expression in a well-typed program has such a derivation tree; the analyser’s job is to build the tree as it walks the parse tree.

Figure 5.2 shows the derivation. The notation in the figure is the standard one from programming-language theory: each inference step writes premises above a horizontal bar and the conclusion below it, with the rule’s name floating to the right of the bar. A derivation is finite tree of such steps, with leaves applying axioms (rules with no premises) and the root being the conclusion about the whole expression.

The shape should look familiar. A typing derivation tree has exactly the same structure as the parse tree the parser just built: leaves are axioms, just as leaves in the parse tree are terminals; interior nodes are inference steps, just as interior nodes in the parse tree are productions. The analyser is, in a precise sense, running a second recogniser over the first recogniser’s output — this one checking not whether the tokens form a sentence, but whether the sentence has a proof. The analyser builds the derivation on the fly, in post-order, attaching the conclusion type to each parse-tree node as it returns from the recursive call.

The rules themselves are the analyser’s specification. We are not going to write down all of them — there are roughly thirty, mirroring the parser’s grammar productions one for one — but the flavour is captured by a handful of representative cases.

The typing rules below use the judgment $\Gamma \vdash e : \tau$ to mean “in context Γ , the expression e has type τ ”. The relation $\tau' \prec \tau$ is the implicit-promotion relation defined in Table 5.1; it is reflexive ($\tau \prec \tau$ for all τ) and includes the two upward promotions $\text{char} \prec \text{int32}$ and $\text{int32} \prec \text{float}$.

Constants. Integer and character literals always have type `int32` and `char`:

$$\frac{}{\Gamma \vdash n : \text{int32}} \quad (n \in \mathbb{Z}) \quad \frac{}{\Gamma \vdash c : \text{char}} \quad (c \text{ a char literal}).$$

Variables. Look up the name in the scope stack:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}.$$

Binary arithmetic on `int32`. Both operands type to `int32`, result is `int32` (analogous rules govern `float`-typed operands and mixed-promotion cases):

$$\frac{\Gamma \vdash e_1 : \text{int32} \quad \Gamma \vdash e_2 : \text{int32}}{\Gamma \vdash e_1 + e_2 : \text{int32}}.$$

Comparison. Both operands must have a common numeric type τ after promotion (or be compatible pointers); the result is `int32` — 0 for false, 1 for true, following C:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \sqcup \tau_2 \in \{\text{int32}, \text{char}, \text{float}\}}{\Gamma \vdash e_1 < e_2 : \text{int32}}.$$

Assignment. Left operand must be an lvalue, not const, with a type that accepts the right-hand side’s type after implicit promotion:

$$\frac{\Gamma \vdash \ell : \tau \quad \ell \text{ lvalue} \quad \neg \text{const}(\ell) \quad \Gamma \vdash e : \tau' \quad \tau' \prec \tau}{\Gamma \vdash \ell = e : \tau}.$$

Function call. Each argument’s type must accept the corresponding parameter type:

$$\frac{\Gamma \vdash f : \tau_1 \times \dots \times \tau_k \rightarrow \tau_0 \quad \Gamma \vdash e_i : \tau'_i \quad \tau'_i \prec \tau_i \quad \forall i}{\Gamma \vdash f(e_1, \dots, e_k) : \tau_0}.$$

The rules are read top-down when stating, bottom-up when applying. To say “what type does $e_1 + e_2$ have”, the analyser first types e_1 and e_2 , then consults the matching rule to combine the results. If no rule matches — because the operands have non-arithmetic types, or the assignment’s left-hand side is an rvalue — the analyser emits a diagnostic and the program is rejected. The inductive structure of the rules mirrors the inductive structure of the parse tree, which is why the analyser can implement them by a post-order visitor: by the time the analyser arrives at a node, every sub-expression’s type is already known.

Algorithm 5.2 states the expression type-checker as a single recursive function with one case per syntactic family. Reading it side-by-side with the inference rules above is the cleanest way to see how the rules turn into code.

Table 5.1 summarises the promotion relation. The arrows go strictly in one direction — toward wider numeric types — and pointers participate only in the explicit-cast world. The distinction between the two columns is exactly the distinction between “the analyser inserts the conversion silently” and “the programmer must write the conversion explicitly with a cast expression”. The relevant Java helpers — `CastCategoryUtil` and `TypePromotion`, both under `semantics/types/` — are consulted by every rule that combines values of potentially different types.

The lvalue/rvalue distinction we met informally in Section 5.1 now has a precise role in the rules: every typing judgment carries, in addition to the type, a *category* attribute marking whether the expression denotes a location or a value. Variable references are lvalues; array indexing is an lvalue; the unary dereference `*p` is an lvalue. Literals are rvalues; function calls (in this subset) are rvalues; the result of `a + b` is an rvalue. The assignment rule above demands that its left operand have lvalue category, which is what rejects `5 = x`: the literal `5` is an rvalue, and no derivation exists. The C standard [40] carries the same machinery under the names “modifiable lvalue” and “object designator”; Kernighan and Ritchie’s classic introduction [43] gives a more readable account that maps closely onto what the FRIScc analyser does.

Key insight

A type system is a small calculus of inference rules, one per syntactic family, that derives the type of every well-formed expression bottom-up. The analyser is the proof-checker: at each node of the parse tree, it tries to apply the rule whose conclusion matches the node’s shape, recursively producing derivations for the premises. If it succeeds, the node is decorated with the derived type and the parent rule fires. If it fails, the program is rejected with a diagnostic locating the rule that did not apply. Every diagnostic the analyser ever emits corresponds to a missing rule application.

Algorithm 5.2: Recursive type derivation for an expression node. The handler dispatches on the node’s kind and recurses into sub-expressions to derive their types before combining them according to the rule for this node. The return value is the derived type, also attached to the parse-tree node as a side effect.

Input: a typing context Γ ; an expression node e in the parse tree

Output: the derived type τ , or a diagnostic if no rule applies

```

1 function TypeOf( $\Gamma$ ,  $e$ );
2   switch kind of  $e$  do
3     case integer literal  $n$  do
4       |   return int32;
5     case character literal  $c$  do
6       |   return char;
7     case identifier  $x$  do
8       |   return Lookup of  $x$  in  $\Gamma$ ;
9     case binary op  $e_1$  op  $e_2$  do
10      |   recursively derive  $\tau_1$  for  $e_1$  and  $\tau_2$  for  $e_2$ ;
11      |    $\tau \leftarrow \text{commonNumeric}(\tau_1, \tau_2)$ ;
12      |   if  $\tau = \perp$  then
13      |     |   report “incompatible operand types”;
14      |   return  $\tau$ ;
15     case comparison  $e_1 < e_2$  do
16      |   recursively derive  $\tau_1$  for  $e_1$  and  $\tau_2$  for  $e_2$ ;
17      |   require  $\text{commonNumeric}(\tau_1, \tau_2) \neq \perp$ ;
18      |   return int32;
19     case cast to  $\tau$  of  $e_0$  do
20      |   recursively derive  $\tau_0$  for  $e_0$ ;
21      |   require the cast  $\tau_0 \rightarrow \tau$  is in the cast-category table;
22      |   return  $\tau$ ;
23     case call  $f(e_1, \dots, e_k)$  do
24      |   look up  $f$  in  $\Gamma$  as  $\tau_1 \times \dots \times \tau_k \rightarrow \tau_0$ ;
25      |   for each argument  $e_i$  from  $i = 1$  to  $k$  do
26      |     |   derive  $\tau'_i$  for  $e_i$ ;
27      |     |   require  $\tau'_i \prec \tau_i$ ;
28      |   return  $\tau_0$ ;

```

From	To	Mechanism
char	int32	implicit sign-extend (8 to 32 bits)
int32	float	implicit Q16.16 lift via <code>itof</code>
float	int32	<i>explicit cast only</i> (<code>ftoi</code> , truncates)
int32	char	<i>explicit cast only</i> (truncates to 8 bits)
$\text{ptr}\langle\tau\rangle$	$\text{ptr}\langle\tau'\rangle$	<i>explicit cast only</i> (no implicit pointer conversions)
τ	void	rejected; void is not an expression type

Table 5.1: The implicit-promotion relation $\prec$ and the explicit-cast table. The upward chain $\text{char} \prec \text{int32} \prec \text{float}$ admits silent widening; the reverse directions and the pointer conversions require an explicit cast, which the analyser checks against the cast-category table in `CastCategoryUtil`. The choice is stricter than ISO C: we forbid implicit `int32`-to-pointer and pointer-to-pointer conversions that ISO C permits, because every silent conversion is one the IR generator and the codegen would later have to second-guess.

Types are not the only contract the analyser enforces. A few static invariants are about *shapes of control flow* or *kinds of declaration*, and they do not fit neatly into a typing judgment. We have to handle them separately.

5.5 Other invariants: loops, returns, and const

Three contracts from the third example in [Section 5.1](#) deserve their own treatment, because each is more about *where* a construct appears than about *what type* it has.

Loop context. The keywords `break` and `continue` have meaning only inside an enclosing loop. `break` jumps to the statement after the loop; `continue` jumps to the next iteration. Outside any loop the keywords have nowhere to go, and the analyser must reject them. The contract is purely positional: a `break` node is fine if and only if the chain of ancestors in the parse tree includes some `<naredba_petlje>` (loop statement) node. FRISCCc’s `JumpStatementRules` tracks this with a *loop-depth counter* on the analyser itself, incremented on entry to any loop node and decremented on exit. We have seen this technique before — a balanced-parenthesis checker uses the same counter. Push on open, pop on close, error if zero when you expected above zero. The only difference here is that “open” is entering a loop and “close” is leaving one. When the analyser visits a `break` or `continue`, it checks the counter; zero means error.

Return paths. Ideally a non-void function should return on every path through its body. The third program from [Section 5.1](#) had a path — the one where the loop finishes naturally — that falls off the end without returning. FRISCCc’s analyser does *not* currently enforce this contract: the program above passes semantic analysis cleanly. We describe the check

here because it is a small, satisfying addition — it is the subject of 5.1 and of the companion project — and because the shape of the check is instructive. The idea is a simple structural property: compute, for each statement subtree, whether it *definitely returns* on every path. The leaves are `return` (yes) and everything else (no); a compound statement returns if any of its statements returns; an `if/else` returns only if both branches return; a `while` or `for` loop never definitely returns (the body might not execute even once). The function-body level then requires that the body’s outermost compound statement definitely returns. Such a check is necessarily conservative — it would flag programs that in fact never reach the end-of-function, like `while (1) { ...}` without `break` — because proving non-termination of arbitrary loops is undecidable. Building exactly that conservative check is what 5.1 asks you to do.

Const-qualified lvalues. A declaration like `const int x = 5;` introduces `x` as an lvalue — it designates a location — but a non-modifiable one. The assignment rule of Section 5.4 carries the predicate $\neg\text{const}(\ell)$ for exactly this reason: even if both type and category check, assignment to a const-qualified lvalue is rejected. The `const` flag travels with the symbol-table entry, set at declaration time and checked at every assignment.

Main signature. FRISCCc’s `GlobalConstraintVerifier` enforces two program-wide invariants in addition to the per-statement checks. The first is that every program declares a `main` function that takes no parameters and returns either `int` or `float`. The empty parameter list is non-negotiable, but the return type is deliberately not pinned to `int` alone, and the reason is architectural rather than lax. A FRISCCc program’s result is simply whatever ends up in register `R6` when the machine halts, which is the value the FRISC simulator and the test harness read back. There is no separate calling convention for returning a `float` — the bits in `R6` are the bits in `R6` — so permitting a `float`-returning `main` is the only semantically honest way to let a program surface a fixed-point result. It is a considered exception driven by the target, not a loose check. The second invariant is that every function *declared* anywhere in the program is also *defined*: a forward declaration with no matching body is a top-level error. Both checks run once, at the end of analysis, against the global scope, which is why they live in their own verifier class rather than in the per-rule visitors.

Algorithm 5.3 ties the four invariants together into one recursive walk. The loop-depth counter is threaded as an explicit parameter, the return-type expectation is another, and the boolean result implements the definitely-returns predicate by structural recursion.

Engineering tip

These four invariants illustrate a pattern worth absorbing: not every static check is a typing rule. Type rules are local — they look at a node and its children — and they compose by structural induction. Positional and global checks need either a moving piece of analyser state (the loop-depth counter, the `const` flag), a small inductive property over subtrees (the definitely-returns predicate), or a program-wide sweep at the end (the main-signature check). A real analyser is a typing calculus plus a few of

Algorithm 5.3: Type-checking a statement with control-flow context: the loop-depth counter d guards `break` and `continue`, τ_R guards `return`, and the boolean result threads the definitely-returns predicate up the tree.

Input: context Γ ; statement s ; loop depth d ; return type τ_R

Output: a “definitely returns” flag; side-effects on Γ and d

```

1 function TypeStmt( $\Gamma, s, d, \tau_R$ );
2   switch kind of  $s$  do
3     case declaration  $x : \tau$  with optional init  $e$  do
4       insert  $x : \tau$  in the top frame; if  $e$  present require  $\text{TypeOf}(\Gamma, e) \prec \tau$ ;
5       return false;
6     case if/else:  $s_1, s_2$  do
7        $r_1 \leftarrow \text{TypeStmt}(\Gamma, s_1, d, \tau_R)$ ;  $r_2 \leftarrow \text{TypeStmt}(\Gamma, s_2, d, \tau_R)$ ;
8       return  $r_1 \wedge r_2$ ;
9     case while or for with body  $s_0$  do
10      TypeStmt( $\Gamma, s_0, d + 1, \tau_R$ );
11      return false (loop may never execute);
12    case return  $e$  do
13      if  $e$  present require  $\text{TypeOf}(\Gamma, e) \prec \tau_R$ ;
14      return true;
15    case break or continue do
16      if  $d = 0$  report “not inside a loop”;
17      return false;
18    case compound  $\{s_1, \dots, s_n\}$  do
19      push a fresh frame onto  $\Gamma$ ;  $r \leftarrow \text{false}$ ;
20      for each  $s_i$  in order do
21         $r \leftarrow r \vee \text{TypeStmt}(\Gamma, s_i, d, \tau_R)$ ;
22      pop the frame from  $\Gamma$ ;
23      return  $r$ ;

```

these auxiliary disciplines, threaded through the same visitor.

The analyser has now done everything it can without leaving the parse tree. What we need to look at next is how it does it: the visitor pattern, the rule families, and the post-order discipline that ties the whole pass together.

5.6 Implementation in FRISCcc

We built the analyser to mirror the shape of its own rules. The analyser lives under `semantics/` in the compiler tree, and its architecture is a one-to-one echo of the rule families. The entry point, `SemanticAnalyzer`, orchestrates the pass. The workhorse, `SemanticChecker`, holds the symbol table, the current loop depth, and the current return-type expectation, and dispatches each parse-tree node to the right handler. We partitioned the handlers by syntactic family into fourteen classes, one per family of non-terminals — binary expressions, control flow, declarations, declarators, expressions, initialisers, jump statements, parameters, postfix expressions, primary expressions, statements, structs, type specifications, and unary expressions — each carrying the suffix `Rules` on its class name. We grouped them this way because the grammar’s non-terminals also fall into roughly the same fourteen families, and the one-to-one mapping makes the analyser’s source navigable from the rules and vice versa. Each class registers its handlers in its own constructor, so the dispatcher only ever consults a single table.

Before any rules fire, the parse tree is converted into a richer *semantic tree*. The conversion is a structural copy — the shape is preserved verbatim — but each node is upgraded to carry fields for the type, the value category, the const flag, and (for identifiers) the resolved symbol. The analyser does not rewrite the tree; it only fills in the fields. We call this the “tree decorator, not tree rebuilder” design, and it pays off later: every downstream pass operates on the decorated tree directly.

The walk itself is *post-order*: the analyser visits a node’s children first, lets their attributes settle, then applies the node’s own rule. The order is dictated by the rules themselves — the binary-arithmetic rule needs the types of its two sub-expressions before it can fire, and “needing the types first” is precisely what post-order traversal supplies. Spelt out as pseudocode, the visitor is half a screen of code.

[Algorithm 5.4](#) captures the discipline. A small fragment of the analyser’s traversal over a loop header makes it visible.

[Figure 5.3](#) shows the visit order on the small expression `t = a + b`. The numbers are the order in which the rule handlers are called. The identifier rule fires three times (once for each leaf, decorating them with their looked-up types). The plus rule fires once (combining the two integer children). The assignment rule fires last (checking the categories and the promotion, then settling the overall type of the assignment expression). Each rule application uses information set by earlier applications and nothing more.

Algorithm 5.4: The post-order visitor that drives the analyser. Children are visited before the node’s own handler runs, so every sub-expression’s type is already attached by the time a parent rule fires. `try/finally` brackets the scope and loop counters in the Java implementation so that `pop` and `decrement` happen even on exception.

Input: a node n in the parse tree; the analyser state (typing context Γ , loop depth d , return type τ_R)

Output: n is decorated with its derived attributes

```

1 procedure Visit( $n$ );
2   if  $n$  opens a new scope (compound, function body) then
3   |   push a fresh frame onto  $\Gamma$ ;
4   if  $n$  opens a new loop then
5   |    $d \leftarrow d + 1$ ;
6   for each child  $c$  of  $n$  from left to right do
7   |   Visit( $c$ ) (recurse before acting on  $n$ );
8   let handler  $\leftarrow$  rulesTable[kind of  $n$ ];
9   invoke handler on  $n$ , attaching type, category, and symbol attributes;
10  if  $n$  closes the loop opened above then
11  |    $d \leftarrow d - 1$ ;
12  if  $n$  closes the scope opened above then
13  |   pop the top frame from  $\Gamma$ ;

```

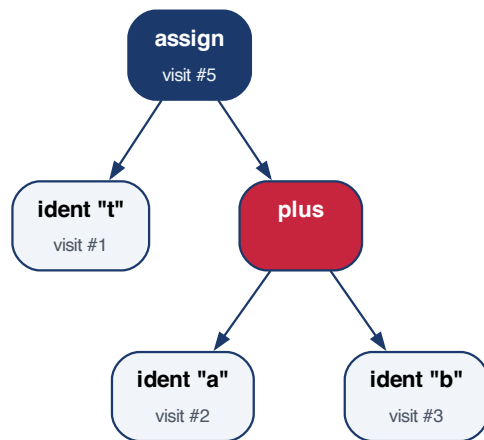


Figure 5.3: A fragment of the parse tree for `for (i = 0; i < n; i++) { t = a + b; ...}` from the loop body of the anchor program. Each interior node is annotated with its post-order visit number. The leaves `a` and `b` receive their types from the variable-lookup rule before the analyser arrives at the parent `+` node; the `+` node combines them via the binary-arithmetic rule and hands its type up to the assignment node; the assignment node, with its `lvalue`-typed left child `t` and arithmetic-typed right child, fires the assignment rule and decorates the parse tree's expression node. The visit numbers are exactly the order in which the rule handlers are called: children before parents, left before right.

The rule classes themselves are short. Each handler does four things, in this order: recurse into the children to settle their attributes; consult the relevant inference rule for this node; record the type, category, and any constant-value attribute back on the node; and emit a diagnostic if any premise fails. A diagnostic halts the pass immediately — the analyser reports the first offending production, in the PPJ-FER style, and throws a `SemanticException`, so the user sees one error at a time rather than a batch. The rule for binary expressions, for instance, is a roughly fifty-line method that does exactly the four steps; the rule for declarations is slightly longer because it has to interact with the symbol table to record the binding. Reading the source of `BinaryExpressionRules` side by side with the binary-arithmetic rule of [Section 5.4](#) is the cleanest way to see how the mathematical specification turns into Java.

Key insight

The analyser is a recursive walk over the parse tree that decorates each node with the attributes the IR generator will read. Children before parents, left before right, types before categories, locals before globals. The visitor pattern encodes the recursion in Java; the post-order traversal encodes the inductive structure of the rules; the fourteen rule classes encode the rule families one-to-one. Everything the analyser does at run time is one of those four nested loops, running in time linear in the size of the parse tree.

The implementation has been described in the abstract. Now we should watch it work on the program we have been carrying around since [Chapter 3](#).

5.7 Worked example: typing `math_fibonacci_iter`

The anchor we tokenised in [Chapter 3](#) and parsed in [Chapter 4](#) is now ready to be typed. We will walk through the analyser's actions in the order they happen — which, because the walk is post-order, is mostly the order of the source lines, with each block's contents finishing before the block itself is decorated.

```
3  int main(void) {
4      int n = 20;
5      int a = 0;
6      int b = 1;
7      int i;
8      int t;
9
10     for (i = 0; i < n; i++) {
11         t = a + b;
12         a = b;
13         b = t;
14     }
15 }
```

```

16     return a;
17 }

```

The analyser begins by visiting the program root. It sees one external declaration, the function definition for `main`. Before walking into the function body, it processes the function declarator `int main(void)`: the return type is `int32`, the parameter list is empty (the keyword `void` marks an empty list in the C subset, not a parameter of type `void`), and the resulting function signature `void → int32` is inserted into the global scope under the name `main`. The function-name binding lives in the bottom of the scope stack; later calls — if there were any — would resolve to it.

Now the body. The analyser opens a function scope on top of the global scope; the function scope is empty in this program because `main` has no parameters. On top of the function scope, the analyser opens block 1 for the function body, and enters the compound statement. The block's declarations are processed first, in source order:

Take `int n = 20`; as the representative case: the declarator records `n : int32` as a new binding in block 1, the initialiser `20` types to `int32`, and the assignment-compatibility check `int32 < int32` succeeds. The next three (`a`, `b`, `i`) follow the same path, and the final `int t`; differs only in that it has no initialiser, which is fine for a local — the analyser records the binding and moves on.

Block 1 now has the five `int32`-typed bindings shown in [Figure 5.1](#). That figure is a snapshot of a *transient* runtime instant: block 1's frame exists only while the analyser is inside the function body, and is popped when the body closes, so it does not survive into the end-of-run dump. The analyser moves on to the statements.

The first statement is the `for` loop. The analyser increments its loop-depth counter from 0 to 1 (so that any nested `break` or `continue` will pass the check) and visits the three clauses of the header in turn. The init clause `i = 0` is an assignment expression: left side is `i`, looked up in block 1 as `int32`-typed lvalue, non-const; right side is `0`, type `int32`; `int32 < int32` holds; the assignment rule fires with result type `int32`. The guard clause `i < n` is a comparison: left is `i` as `int32`, right is `n` as `int32`, common type is `int32`, the comparison rule fires with result type `int32` (0 or 1). The update clause `i++` is the post-increment on an `int32`-typed lvalue, which itself types to `int32`.

Now the body. The analyser opens block 2 for the `for`-loop body, on top of block 1. Block 2 is empty (no declarations) but it sits on the stack for the duration of the body's statements. The three assignments inside the body are processed in order. `t = a + b`; has been analysed all the way through in the previous sections; [Figure 5.2](#) is its derivation tree, and the assignment rule decorates the whole expression with type `int32`. `a = b`; is the simplest possible assignment: both sides are `int32`-typed lvalues (well, `b` is an lvalue but its value is read), `int32 < int32`, accepted. `b = t`; is analogous. When the closing brace of the `for`-body arrives, the analyser pops block 2 off the stack, leaving only block 1 (and the function and global scopes below it). The loop-depth counter drops back to 0.

The last statement of `main` is `return a;`. The analyser consults the current return-type expectation, which was set to `int32` when block 1 was opened on top of the `int`-returning `main`. It types the expression `a` (lookup in block 1 gives `int32`) and applies the return rule: `int32 < int32` holds, so the return is accepted.

The closing brace of the function body pops block 1; the closing brace of the file pops the function scope; the analyser is back at the global scope with one binding (`main`). The `GlobalConstraintVerifier` now runs: it looks up `main` in the global scope, finds its signature, checks that it takes no parameters and returns either `int` or `float` (both an `int`- and a `float`-returning `main` with an empty parameter list pass), confirms that every declared function is also defined, and passes. No diagnostic has been emitted anywhere; the program is well-typed.

The artifact of the pass is the decorated parse tree, written to `compiler-bin/semantic_tree.txt` when the user runs `./run.sh --sem math_fibonacci_iter`. The file lists every non-terminal and terminal node with its source location and any attribute the analyser attached: type, category, const flag, resolved symbol — so the relational node for `i < n`, for instance, shows `type=int`. The symbol-table section that follows reflects the *end* of the run rather than its peak: because the local frames are popped as each block closes, it reports only the surviving global binding for `main` and shows the function-body scope as empty. The transient block-1 frame of Figure 5.1 lived only while the analyser was inside the body. (The book’s listings copy is regenerated from the compiler by the `/regen-anchor` command, so any drift between the analyser and the prose is caught the next time the anchor is refreshed.) The IR generator of Chapter 6 reads this file directly and never has to re-derive the type of any expression.

Engineering tip

The most efficient way to debug a semantic-analyser problem is to dump the decorated tree and compare its attributes against your hand-derived expectations. FRISCCc’s report format is designed to be read top-to-bottom in source order: each line shows one node, its kind, its attributes, and its position. When a type is wrong, the wrong type is visible at the node that wears it, and tracing backward through the node’s children usually finds the rule that misfired in a handful of steps.

Soundness. We would like to know that any program our analyser blesses cannot “go wrong” at run time — that the prediction made by every type label actually holds when the program executes. This is the *type soundness theorem*, proved by the progress + preservation technique due to Wright and Felleisen [86] and given its modern textbook treatment by Pierce [69]. Their contribution was to replace the older denotational soundness arguments — which leaned on coherence properties of domain-theoretic semantic functions — with two crisp lemmas stated entirely in terms of small-step reduction. Almost every type-soundness proof published since uses the same two-lemma machinery, and so will ours. The proof proceeds by induction on the typing derivation, after we have the IR’s small-step reduction semantics in hand. The *progress lemma* says that a well-typed, non-final configuration can always step; the *preservation lemma* says that a step from a well-typed configuration

produces a well-typed configuration; their combination gives the soundness theorem. We defer the formal statement and proof to [Appendix G](#) for a technical reason: the theorem is a statement *about the IR's operational semantics*, not about the source language, so we need the reduction relation of [Chapter 6](#) before the proof can be written down. The *canonical-forms lemma* — that a value of each type has one of a finite enumerated set of forms — is the auxiliary result the proof leans on hardest, and we sketch it there as well.

For now the moral is enough: the analyser's contract with downstream is not merely a syntactic one. It is the substantive promise that a well-typed source program produces a well-typed IR, which executes without getting stuck. The IR generator of [Chapter 6](#) reads that tree and never has to re-derive a type. Its job is to turn decoration into instruction — and that is exactly where we go next.

Practice

Exercise 5.1 (Applied · ★★☆)

The third program in [Section 5.1](#) had a non-void main that falls off the end of its body on at least one control path — a contract FRISCcc does not currently enforce. State, as precisely as you can, the definitely-returns predicate for an if/else statement and for a while loop that a stricter analyser would use. Why does the predicate's value for while (1) { ...} have to be conservative? Sketch a program that the conservative check rejects but that would never in fact run off the end at run time.

Exercise 5.2 (Applied · ★★☆)

Write a one-line C subset program that the parser accepts but the analyser must reject for each of the following reasons: (a) the left-hand side of an assignment is an rvalue; (b) an identifier is used in a scope where it is not declared; (c) a comparison combines operands of incompatible numeric types; (d) a break appears outside any loop. For each, identify which inference rule (or which auxiliary check) the analyser uses to detect the problem.

Exercise 5.3 (Applied · ★★☆)

The promotion relation $\tau' \prec \tau$ is reflexive and admits two upward steps ($\text{char} \prec \text{int32}$, $\text{int32} \prec \text{float}$). Is it transitive? In particular, does $\text{char} \prec \text{float}$ hold implicitly? Argue your answer from the relation's definition, and then test it on a small program of the form `char c = 'A'; float f = c;;` does FRISCcc accept the program, and why?

Exercise 5.4 (Applied · ★★★)

Hand-simulate the analyser’s symbol-table operations during analysis of the anchor program’s `for` loop. List the `push`, `pop`, `insert`, and `lookup` operations in the order they occur, from entry into the loop header to exit from the loop body. There should be a single `push`, a single `pop`, no `inserts` (the loop introduces no new declarations in this program), and one `lookup` per identifier occurrence in the header and body.

Exercise 5.5 (Applied · ★★★)

Construct, in inference-rule notation, a typing derivation for the expression `(i < n) || (a == 0)` in the FRISCCc type system, assuming `i`, `n`, and `a` are all declared as `int` in the typing context Γ . The derivation has six leaves and interior nodes: two uses of the T-VAR rule (one per comparison operand group), two applications of the comparison-typing rule, and one application of the logical-or rule. Write each step as a fraction with hypotheses above the line, the judgement $\Gamma \vdash e : \tau$ below, and the rule name to the right. Verify that every node in the figure [Figure 5.2](#) follows the same schematic.

Project

Exercise 5.6 (Lab · ★★★)

Goal. Implement the type-checker and scope manager for the `tinyC` subset in the companion repository’s `ch05- semantics/` module.

Background. `tinyC`, defined in `tinyc-spec/SPEC.md`, is the `int`-only fragment of `C`: no floats, no chars, no pointers, no arrays, no structs. The analyser is correspondingly small. It still needs to enforce three contracts: every identifier used in an expression must be declared in some enclosing scope; every assignment’s left side must be an lvalue; and every non-void function must return on every path. Without floats or chars the promotion relation collapses to the identity, so type-checking arithmetic is just “both operands must be `int32`”.

Steps. Open `frisccc-companion/ch05- semantics/starter/src/main/java/com/frisccc/tinyc/ semantics/Analyzer.java`. The starter sketches the visitor with `TODO` markers on each rule family; your job is to fill in the four rule families that `tinyC` needs (declarations, expressions, statements, jumps) and the scope-stack management around them. A working reference implementation lives in `ch05- semantics/solution/`. The tests in `ch05- semantics/tests/` cover both positive cases (well-typed programs the analyser must accept) and negative cases (programs that the analyser must reject with a specific error message).

Acceptance. Run `mvn test -pl ch05- semantics` from the companion root. All eight tests must pass. Four are worth naming explicitly. `WellFormedFibonacciTypeChecks` requires the `tinyC` translation of `math_fibonacci_iter` to analyse without diag-

nostics. `UndeclaredIdentifierFails` requires `int main() { return x; }` to throw a `SemanticError` whose message contains “undeclared identifier: x”. Closely related, `ReturnMissingOnPathFails` expects `int main() { if (1) return 1; }` to throw with the message “may not return on every path”, and `WhileLoopDoesNotCountAsReturn` extends the same expectation to `int main() { while (1) {} }`, because the analyser’s definitely-returns predicate is conservative and a loop body might never execute.

Reflection. Once your code passes, answer in writing: how many lines did your scope-management code take, and which of those lines could you imagine sharing between `tinyC`’s analyser and an analyser for the `FRISCCc` subset? What does the answer suggest about the right place to draw the boundary between subset-specific and subset-generic semantic-analyser code?

Notes and further reading

The progress + preservation technique for proving type soundness is due to Wright and Felleisen [86]. Their 1994 contribution was to replace the older denotational arguments — which leaned on coherence properties of domain-theoretic semantic functions — with two simple lemmas stated in terms of small-step reduction. The technique has dominated soundness proofs ever since. Pierce’s textbook on types [69] is the source most readers should consult first; its Chapter 8 introduces progress and preservation for a small typed arithmetic language, Chapter 9 extends them to the simply typed lambda calculus, and the later chapters extend the pattern to richer type systems. Harper’s practical foundations [34] covers the same material with a more uniform formal style, and is the next stop for readers who want type theory as one coherent framework rather than a sequence of case studies.

`FRISCCc`’s analyser only ever *checks* types — every variable arrives already annotated, so the analyser never has to guess one. The richer problem of *inferring* the type of an unannotated binding, which underlies ML and Haskell, is solved by the Hindley–Milner algorithm, whose principal-type result is due to Hindley [37] and was independently rediscovered and turned into the practical Algorithm W by Milner. We do not need any of that machinery here — our type rules are read off the parse tree directly — but a reader curious about what type-checking looks like when the types are not written down should start with that paper.

For the engineering side — the visitor pattern, the symbol table as a tree of scopes, the rule-family decomposition — the two canonical references are the Dragon Book [2] and Appel’s textbook [7]. The Dragon Book’s Chapter 6 covers intermediate-code generation, with type checking treated in §6.5; Appel’s Chapter 5 takes the same material at a slower pace. Both predate the modern emphasis on formal type soundness, and treat type-checking primarily as an algorithmic-correctness problem; combined with Pierce, they give a complete picture.

The lvalue/rvalue distinction is articulated in detail in the C standard [40], where “object designator” takes the place of the historical term “lvalue”. The classic Kernighan and Ritchie introduction [43] treats the same distinction more readably in Chapter 2, and

FRISCcc's restricted version of the rules follows the K&R treatment more closely than the C17 standard's.

The PLT-FER course [27] on which the FRISCcc parser and analyser are based uses Croatian non-terminal names in its specifications (`<naredba_skoka>`, `<slozena_naredba>`, and so on). The full glossary is in Appendix A; we have used the Croatian names sparingly in this chapter, only where the actual implementation refers to them.

Your turn

The companion repository's `ch05-semantic/` module is your build-along for this chapter. From the companion root, run `cat ch05-semantic/README.md` for the order in which to attack the four rule families. The starter file is `Analyzer.java` under `ch05-semantic/starter/`; the tests — including the bad programs from Section 5.1 — are in `ch05-semantic/tests/`; the reference implementation is in `ch05-semantic/solution/`. Run `mvn test -pl ch05-semantic` when you are ready to check your work. When the tests pass, you have a type-checker. The next chapter turns the decorated tree into intermediate code.

Chapter 6

Lowering to typed IR

“All problems in computer science can be solved by another level of indirection.”
— David Wheeler

At the end of [Chapter 5](#) we held in our hands a typed semantic tree: every identifier resolved to a symbol-table entry, every expression decorated with its type, every implicit conversion made explicit, every `break` hooked up to the loop it escapes from. The tree is, in a precise sense, the meaning of the source program made manifest. Read it carefully and you can answer almost any question about what the program is supposed to do.

What it cannot tell you is how to do it on a machine. The tree is shaped like the source: an `if` is a node with three children, a `for` is a node with four, an expression is a tower of binary operators. There is no notion of basic blocks, no notion of temporaries, no notion of jumps. The FRISC architecture, by contrast, knows about exactly those things and not much else. If we tried to walk the tree and emit FRISC directly, we would be answering ten questions at once: which register holds the loop counter, where the back-edge jumps to, when to spill, how to fold the implicit conversions, how to short-circuit the `&&` on the third line. We would do all of it once, and then redo it the next time we touched the code generator, and again the next, because none of the work would be reusable.

The standard escape from this trap is one of the oldest tricks in our craft: insert a level of indirection. Between the semantic tree and the assembly, we put a third language — a small, deliberate, explicitly-typed notation that has just enough machine-like structure to make code generation mechanical, and just enough source-like structure to keep optimisation tractable. That notation is the *intermediate representation*, and the rest of this chapter is about building it. We will look at why it earns its keep even in a single-source-single-target compiler like FRISCcc, we will define the notation by example, and we will lower our chapter anchor, `math_fibonacci_iter`, into it block by block. By the time we reach the back end in [Chapter 8](#), the IR will feel like the language the compiler actually thinks in.

6.1 Why not generate FRISC directly?

The honest objection comes first. Compiler textbooks usually motivate the IR with the $N \times M$ argument: if you support N source languages and M target machines, a shared IR

saves you from writing $N \times M$ compilers and lets you get away with N front ends and M back ends. The argument is true and it is the reason GCC and LLVM look the way they do. It is also, for us, an academic point. FRISCCc has one source language and one target. The $N \times M$ ledger reads $1 \times 1 = 1 = 1 + 1 - 1$, which is to say: no savings.

So let us look at a small example and see what we actually buy. Take the expression $a + b$, where a and b are local int variables sitting at frame offsets 4 and 8. A direct, tree-walking code generator wants to emit something like

```
LOAD  R1, (R5+4)    ; load a
LOAD  R2, (R5+8)    ; load b
ADD   R1, R2, R0     ; R0 := a + b
```

and it has to do that walk every single time it encounters a binary plus, in every context that addition can appear, with all the register-allocation decisions baked into the walk itself. Replace $a + b$ with $(a + b) * 2 + c$ and the walker has to know when to flush $R0$ into a fresh register before it gets clobbered by the multiply, when the constant 2 should be inlined as an immediate and when it should be hoisted, and what to do if c spills. None of this is hard. All of it is glued to the tree shape.

Now suppose instead that the front end emits an explicit, flat sequence,

```
t1 = addr_of_symbol local:a
t2 = load t1 : int32
t3 = addr_of_symbol local:b
t4 = load t3 : int32
t5 = add t2, t4 : int32
```

and the code generator's job is just to assign $t1\dots t5$ to FRISC registers and translate each line independently. The line count went up. The total work went down, because the work in the front end is now algebraic and uniform: every operation is a single line with a destination, an opcode, and operands of explicit type. We pay in verbosity, we are reimbursed three times over.

First, optimisations live somewhere uniform. [Chapter 7](#) will spend an entire chapter on the sixteen passes FRISCCc applies to the IR. Each pass is a rewriter from IR to IR. None of them would even type-check on the semantic tree, because the tree does not have the right shape: it has expressions inside expressions, control flow glued to syntax, no notion of a basic block to slide a redundant load out of. The IR is the medium in which the optimisations are *expressible*.

Second, the IR is executable by itself. [Chapter 11](#) will give it an interpreter that executes IR instructions one at a time against a tiny abstract machine, with no detour through FRISC assembly. That interpreter is independently useful (it lets a reader of this book run their programs without needing the FRISC simulator) but its real job is to be the reference oracle: any disagreement between `run-ir` and `--run` on the same input is, by construction,

a bug in code generation. The IR earned that property by being machine-independent. It would not have it if we generated FRISC directly.

Third, the back end becomes replaceable. The original CompCert paper [55] makes this point starkly: their formal verification effort is structured around exactly the same separation, and the proofs about IR-level optimisations transfer to any back end that consumes the IR correctly. We will not retarget FRISCcc to ARM in this book. We will not even retarget it to RISC-V. But the option exists, in a way it would not if the front end committed to FRISC at expression-tree time. And it costs nothing more than the IR we are already building.

Three reasons, then: optimisations need somewhere to live, the IR is executable on its own, and the back end is decoupled. The combinatorial argument is the one that gets repeated; these three are the ones that actually justify what we are about to do.

6.2 Three-address code, in five minutes

The fragment we just looked at was three-address code, and it is worth stopping to look at it as a technique on its own, because once you see what it is doing you see it everywhere.

Take a small expression with some nesting:

$$t = (a + b) * 2$$

A tree IR represents it as a tree with $*$ at the root, $+$ as the left subtree, the constant 2 as the right subtree, and identifiers at the leaves. Reading the tree top-down tells us what the expression *is*; it does not tell us anything about the order in which the machine should compute it. To get to something the machine can run, we flatten the tree into a sequence in which every step does exactly one operation, every operation names its inputs and its output, and every name refers to a value:

```
t1 = a + b
t2 = t1 * 2
t  = t2
```

That is three-address code. The name comes from the shape of each instruction: at most three addresses are mentioned — two source operands and one destination — and the operator binds them. Aho, Sethi and Ullman codified the form in the Red Dragon [3] and the Purple Dragon [2] attributes it back to the early 1970s; Cooper and Torczon's *Engineering a Compiler* [18] works it as a running example for hundreds of pages. The reason it has held its place for half a century is the shape itself: every instruction looks the same. Whatever transformation we want to do — constant fold, copy propagate, common subexpression eliminate, hoist out of a loop — it is a rewrite over a uniform sequence rather than a recursion over a heterogeneous tree.

The cost is real. The expression $(a + b) * 2$ became three instructions and introduced two fresh names. A small program typically grows by a factor of three or four when it crosses the boundary into the IR; our anchor program produces twenty-two temporaries from its sixteen lines of C. The optimiser will earn most of them back, but the IR as the front end emits it is verbose by design.

There is a well-known stronger discipline that pushes the same idea further: *static single assignment* form, in which every temporary is assigned exactly once over the whole function. SSA is beautiful, it makes many dataflow problems trivial, and it is what LLVM and GCC's GIMPLE actually use [21, 53]. We have chosen not to use it.

Why FRISCcc is not in SSA. Strict SSA requires that every name be written in exactly one place in the program. The price for that uniqueness, at any point where control flow merges, is a ϕ -function: a virtual instruction that picks the value coming in along each predecessor edge. For an if-then-else or a while loop, ϕ -nodes proliferate quickly, and they require their own dataflow story to reason about.

Production compilers take that price gladly: the optimisations SSA enables — sparse conditional constant propagation, global value numbering, scalar replacement of aggregates — are dramatically easier to implement and dramatically more effective in SSA. For a teaching compiler whose first job is to make the lowering transparent, ϕ -functions are a distraction. Our compromise is to use three-address code with function-scope unique names but without the single-assignment rule: temporaries are mutated freely, slots are reloaded explicitly, and the optimiser handles the redundancy with simpler local rewrites rather than a global dataflow machinery. Readers who want the SSA experience will find [Chapter 7](#)'s `CopyPropagationPass` and `GlobalValuePropagationPass` are doing, in a smaller way, the work that SSA construction would have done up front.

What we have, then, is three-address code with two extra qualities: every value carries an explicit type annotation (more on that in the next section), and the linear sequence is partitioned into basic blocks with explicit terminators. The first quality lets the optimiser reason locally about which rewrites are legal; the second gives it the control-flow graph it needs to know where the loops are. With those two additions in place, we are ready to look at the notation itself.

6.3 Our typed IR

Rather than handing down a grammar and then explaining what its productions mean, we will tour the notation by reading the real output of FRISCcc's IR printer on the chapter's anchor program. The file lives at `book/listings/math_fibonacci_iter/intermediate_00.ir` and we will see it complete in [Section 6.6](#); for now we zoom in on the parts that introduce a new bit of syntax.

The program envelope

Every IR file begins with `.program` and ends with `.endprogram`. In between are zero or more functions, each delimited by `.func` and `.endfunc`. A function header looks like

```
.func main():int32
```

which reads exactly as it looks: the function is called `main`, takes no parameters, and returns an `int32`. There is no space between the closing parenthesis and the colon; this is a deliberate notational choice that makes the return-type annotation visually glued to the function signature.

A function is then divided into two pieces by directive: a frame description and a sequence of basic blocks. The frame description comes first:

```
.frame locals=20 bytes align=4
.slots
  local n@0:int32
  local a@4:int32
  local b@8:int32
  local i@12:int32
  local t@16:int32
```

The `.frame` line is the size and alignment of the local-variable region, in bytes. Five `int32` variables at four bytes each fill exactly twenty bytes, and the frame is aligned to a four-byte boundary because that is what FRISC’s word-addressed loads expect.

The `.slots` table that follows is the precise layout of those twenty bytes. Each line gives a slot *name*, an *offset* inside the frame written with `@`, and a *type*. Slot `n@0` means: the local variable called `n` lives at offset zero in the frame; an offset of zero is the lowest-address slot. The back end will translate `n@0` into an address-mode like `(R5+0)` where `R5` holds the frame pointer; the IR itself does not commit to any particular register.

The slot table is the IR’s only memory model for local variables. Every read and every write of a local goes through it. There is no implicit `a` that you can mention by name in an instruction; you must first ask for the address of `a`’s slot, and then either load from or store to that address. The verbosity is intentional: it makes loads and stores first-class citizens that the optimiser can shuffle and elide, and it makes the memory traffic of a function visible at a glance.

Figure 6.1 shows the same information graphically: slot zero is at the frame pointer, slot four is the next word down, and so on, growing toward higher addresses. Nothing in this picture is FRISC-specific yet — the IR’s slot table is an abstract memory layout that any sensible back end could realise. The fact that FRISC happens to use `R5` as the frame pointer is a back-end choice we will commit to in [Chapter 8](#).

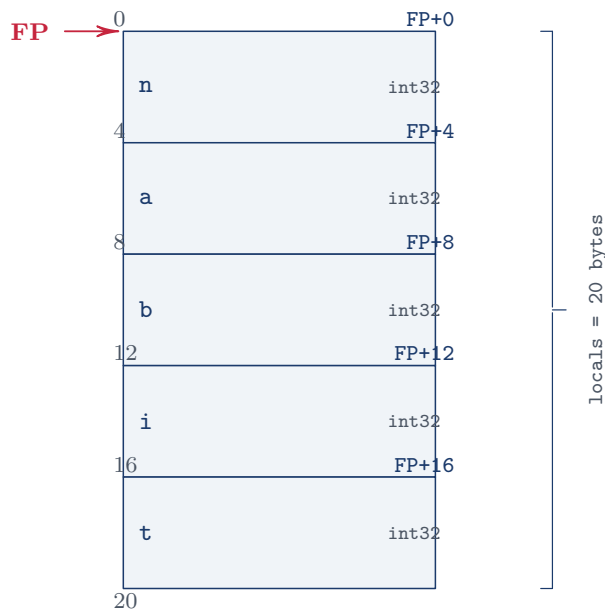


Figure 6.1: The five slots of `main()`'s activation record, laid out by ascending offset from the frame pointer. Each slot is one `int32`; the five together fill exactly the twenty bytes that the `.frame` directive declares. The back end will translate each `name@offset` into a `(R5+offset)` addressing mode at codegen time.

Basic blocks and terminators

After the slots comes the `.blocks` directive and then a sequence of *basic blocks*: maximal straight-line code fragments, each starting with a label and ending with exactly one control-flow instruction. Our anchor produces five of them, named `L0` through `L4`. A block looks, in skeleton, like

```
L1:
    t4 = addr_of_symbol local:i
    t5 = load t4 : int32
    t6 = addr_of_symbol local:n
    t7 = load t6 : int32
    t8 = cmp_lt t5, t7 : bool
    br t8, L2, L4
```

and you can tell where it ends by looking for the terminator: `br`, here, is a conditional branch on `t8` to either `L2` or `L4`. The other terminators are `jmp L` for an unconditional branch and `ret` (or `ret t`) for a return. There is no fall-through. A block that ends without a terminator is malformed; control always leaves a block through a terminator instruction, never by running off the bottom.

This is one of those design choices that costs us a little verbosity and earns us a great deal of structural clarity. Fall-through is implicit; explicit jumps are not. The optimiser can rearrange blocks freely without worrying about which block was lexically next; the code generator can lay blocks out in whatever order minimises branch distance; the verifier can confirm in a single scan that every block ends correctly. Compiler engineers who have spent time chasing bugs in fall-through assembly will recognise the trade as well worth making.

The right-hand side algebra

Inside a block, every non-terminator instruction has the shape `t = op operands : type`: a destination temporary on the left, an opcode and its operands in the middle, and the result type on the right. The opcode is drawn from a small fixed catalogue summarised in [Table 6.1](#). The operand types must be consistent with the opcode (`add` requires two operands of the same arithmetic type; `cmp_lt` returns `bool` no matter what its operands are), and the result type on the right of the colon is what the operation produces.

A few entries deserve a sentence. Address computation has three opcodes because there are three kinds of l-value in our source language: a plain variable becomes `addr_of_symbol`; an array element becomes `addr_index` of a base, an index, and the element size in bytes; a struct field becomes `addr_field` of a base and a `Struct.field` name. All three return a pointer, and the pointer is then either loaded or stored. The `store` opcode is the only IR instruction besides terminators that does not have a destination temporary, because storing a value to memory does not produce a value: its written form is `store addr, val : type`, with the address first, the value second, and the type tail naming the value's type.

Category	Opcodes	Notes
Arithmetic	add, sub, mul, div, mod	integer or fixed-point
Bitwise	and, or, xor, shl, shr	integer only
Comparison	cmp_eq, cmp_ne, cmp_lt, cmp_le, cmp_gt, cmp_ge	result is bool
Unary	neg, not	negate; bitwise complement
Address	addr_of_symbol, addr_index, addr_field	slot; base, idx, elemSize; base, Struct.field
Memory	load, store	store takes addr, val
Casts	trunc, sext, zext, itof, ftoi	width, sign, int/float conversion
	ptrcast	pointer reinterpretation
Calls	call	destination temporary receives the return value
Inc/dec	preinc, postinc, predec, postdec	operand is an address; result is new/old value

Table 6.1: The right-hand side opcodes of FRISCCc’s IR. Every non-terminator instruction picks an opcode from this table, supplies operands of consistent type, and annotates the result type after the colon.

Constants are written with a leading # and a trailing type annotation: `#20:int32`, `#1:int32`, `#0.5:float`, `null:ptr<int32>`. The hash is the IR’s syntactic marker for “literal value, not a name”; the type annotation is mandatory, because the optimiser needs to know whether `#0` is a 32-bit zero, a fixed-point zero, or a pointer null. There is no implicit type. Even Boolean literals are written `#true:bool` and `#false:bool`.

The type catalogue that those constants and result annotations are drawn from is small (Table 6.2). It maps closely onto the source-language types but it is not identical: the source has `int`, the IR has `int32`, and the difference matters because the IR commits to a precise machine width while the source leaves it abstract. The source `float` stays `float` in the IR type catalogue; the difference is one of *realisation* rather than name, because FRISC has no hardware floating-point and the back end realises every IR `float` as Q16.16 fixed-point with runtime helpers. The IR type token is still `float`.

IR type	Meaning
<code>int32</code>	32-bit signed integer (the C <code>int</code>)
<code>char</code>	8-bit signed integer (the C <code>char</code>)
<code>uchar</code>	8-bit unsigned integer (the C <code>unsigned char</code>)
<code>float</code>	the C <code>float</code> ; realised as 32-bit Q16.16 fixed-point
<code>bool</code>	1-bit logical value (comparison results)
<code>void</code>	no value (functions that do not return one)
<code>ptr<T></code>	pointer to a value of type <code>T</code>
<code>array<T,N></code>	contiguous array of <code>N</code> values of type <code>T</code>
<code>struct{...}</code>	record type with named, typed fields

Table 6.2: The IR’s type catalogue. Every value flowing through an IR instruction has one of these types; every operation is well-formed only when its operand and result types agree.

The catalogue is intentionally smaller than C's: no short, no long, no unsigned int. This is the same deliberate narrowing that bytecode VMs like the JVM make — the IR's type system is sized to what the optimiser can reason about uniformly, not to what the source language can express.

With slots, blocks, terminators, opcodes, constants, and types in hand, we have read the entire notation. What is left is the mechanical question of how a typed semantic tree turns into IR text that obeys all of those constraints, and that is where we go next.

6.4 Lowering: a recursive descent through meaning

The lowering walks the typed tree from [Chapter 5](#) and produces IR. It is the most straightforward pass in the compiler if you frame it correctly. Frame it incorrectly and it is the most fragile.

Frame it correctly: think of lowering as a structural recursion where each call returns two things — the IR instructions emitted so far, accumulated into the current basic block, and a *value handle* naming where the result of the expression ended up. A value handle is usually a temporary, sometimes a constant, occasionally an address. The caller decides what to do with the handle: store it into a slot, feed it as an operand to a bigger expression, hand it to a comparison.

Let us walk four cases by hand. They are not the full case analysis — the source language has many more nodes — but they are representative, and after four cases the pattern is clear.

Lowering a literal. The source contains 20. The semantic tree node is an integer-literal node carrying `value=20` and `type=int32`. Lowering emits no instructions and returns the value handle `#20:int32`. That is the entire case: literals are their own handle.

Lowering an identifier in r-value position. The source contains `i` on the right of a comparison. The semantic tree node is an identifier-expression node carrying `symbol=local:i` and `type=int32`. Lowering allocates two fresh temporaries, emits

```
t4 = addr_of_symbol local:i
t5 = load t4 : int32
```

and returns the value handle `t5`. Every identifier read in the IR is exactly this pair: first take the address of the slot, then load through it. The redundancy — if `i` is read twice in adjacent statements we will compute its address and load it twice — is deliberate: the optimiser will collapse it in [Chapter 7](#), and meanwhile the IR remains transparent.

Lowering a binary expression. The source contains `a + b`. The semantic tree node is a binary-expression node with `op=PLUS`, a left subtree, a right subtree, and `type=int32`. Lowering recursively lowers the left operand to a handle h_l , recursively lowers the right operand to a handle h_r , allocates a fresh temporary t , and emits `t = add  $h_l$ ,  $h_r$  : int32`. It returns t . The case is short because the heavy lifting was already done by the recursive lowering of the operands.

Lowering an assignment. The source contains `a = b`. The semantic tree node is an assignment-expression with an identifier on its left-hand side naming `local:a` and an arbitrary subtree on its right. The right-hand side gets lowered as an r-value, producing a value handle h . The left-hand side gets lowered as an *l-value*, which is a different mode of the same walker: it emits an `addr_of_symbol` (or `addr_index` or `addr_field`) and returns the resulting pointer handle p . Lowering then emits `store  $p$ ,  $h$  : int32`, and the assignment is complete. The two modes — r-value and l-value — are why our IR walker is a pair of mutually recursive functions rather than a single function; the distinction matters for any expression that can sit on the left of `=`, namely identifiers, array indexing, struct field access, and pointer dereference.

Key insight

Two walkers, one tree. The lowering has two modes: an *r-value mode* that returns a temporary (or constant) naming the computed value, and an *l-value mode* that returns a pointer to a slot. Every expression that can appear on the left of `=` — identifier, array subscript, struct field, pointer dereference — must be handled by both modes. An expression that cannot appear on the left (a literal, a binary operation, a function call) needs only the r-value mode. The distinction is what makes `t = a + b` and `a = t + b` lower correctly without duplicating the address-of-symbol logic.

The pattern across the four cases generalises into the recursive walker that [Algorithm 6.1](#) captures. The walker is called once per expression node, returns a value handle, and emits zero or more IR instructions into the current block as a side effect. The l-value mode ([Algorithm 6.2](#)) is the same recursion with a different return contract — it returns a pointer rather than a value — and a strictly smaller case analysis, because only the four l-value-eligible node kinds need to handle it.

[Algorithm 6.1](#) is the r-value half of the value-handle pattern. Reading the cases top to bottom is reading the shape of the language's expression syntax: literals are their own handle, an identifier read produces a load through the address-of-symbol intermediary, a binary operator emits a single three-address instruction over the recursive handles of its sub-expressions, and the two address-of/dereference cases are the bridge to the l-value walker. The recursion is linear in the size of the expression: each node produces a constant number of IR instructions, no node is visited twice, and the value handle threads upward through the tree exactly once. The l-value walker below mirrors the same structure on the other side of the bridge.

Algorithm 6.1: Expression lowering to a temporary (r-value mode). Each case emits at most a constant number of IR instructions and recurses on direct children; the total work is linear in the size of the source expression.

Input: a typed expression node e with type τ , the current block builder B , and the symbol environment Σ

Output: a value handle h naming the result of e , with all generated instructions appended to B

```

1 switch kind of  $e$  do
2   case integer or boolean literal with value  $v$  do
3     return  $\langle v, \tau \rangle$  (constants are their own handle)
4   case identifier read of symbol  $s$  do
5      $p \leftarrow \text{newTemp}(); \text{emit}(p = \text{addr\_of\_symbol } s)$ 
6      $t \leftarrow \text{newTemp}(); \text{emit}(t = \text{load } p : \tau)$ 
7     return  $t$ 
8   case binary op  $e_1 \odot e_2$  do
9      $h_l \leftarrow \text{lowerExpr}(e_1); h_r \leftarrow \text{lowerExpr}(e_2)$ 
10     $op \leftarrow \text{IR opcode for } \odot$  (one of add, sub, mul, div, mod, and, or, xor, shl, shr)
11     $t \leftarrow \text{newTemp}(); \text{emit}(t = op \ h_l, h_r : \tau)$ 
12    return  $t$ 
13  case comparison  $e_1 \sim e_2$  with relation  $\sim$  do
14     $h_l \leftarrow \text{lowerExpr}(e_1); h_r \leftarrow \text{lowerExpr}(e_2)$ 
15     $op \leftarrow \text{cmp opcode for } \sim$  (one of cmp_eq, cmp_ne, cmp_lt, cmp_le, cmp_gt, cmp_ge)
16     $t \leftarrow \text{newTemp}(); \text{emit}(t = op \ h_l, h_r : \text{bool})$ 
17    return  $t$ 
18  case cast of  $e_0$  to  $\tau$  do
19     $h \leftarrow \text{lowerExpr}(e_0); t \leftarrow \text{newTemp}()$ 
20     $op \leftarrow \text{cast opcode by source and target widths}$  (one of trunc, sext, zext, itof, ftoi)
21     $\text{emit}(t = op \ h : \tau)$ 
22    return  $t$ 
23  case call  $f(e_1, \dots, e_k)$  do
24     $h_i \leftarrow \text{lowerExpr}(e_i)$  for  $i = 1 \dots k$ 
25     $t \leftarrow \text{newTemp}(); \text{emit}(t = \text{call func:} f(h_1, \dots, h_k) : \tau)$ 
26    return  $t$ 
27  case address-of  $\&e_0$  do
28    return  $\text{lowerLvalue}(e_0)$  (delegate to l-value walker)
29  case dereference  $*e_0$  do
30     $p \leftarrow \text{lowerExpr}(e_0); t \leftarrow \text{newTemp}()$ 
31     $\text{emit}(t = \text{load } p : \tau)$ 
32    return  $t$ 

```

Algorithm 6.2: Expression lowering to a pointer (l-value mode). The four cases are exactly the four kinds of expression that can appear on the left of an assignment in our source language; any other expression kind invoked in l-value mode is a semantic-analysis bug that should have been caught earlier.

Input: a typed l-value expression node e with type τ , the current block builder B , and the symbol environment Σ

Output: a pointer handle p such that loading from p yields the value denoted by e and storing to p updates it

```

1 switch kind of e do
2   case identifier read of symbol s do
3      $p \leftarrow \text{newTemp}(); \text{emit}(p = \text{addr\_of\_symbol } s)$ 
4     return  $p$ 
5   case array index  $e_b[e_i]$  do
6      $h_b \leftarrow \text{lowerExpr}(e_b)$  (base, evaluated as r-value pointer)
7      $h_i \leftarrow \text{lowerExpr}(e_i)$  (index, evaluated as r-value int32)
8      $p \leftarrow \text{newTemp}(); \text{emit}(p = \text{addr\_index } h_b, h_i, \text{sizeof}(\tau))$ 
9     return  $p$ 
10  case struct field  $e_s.f$  of struct type  $S$  do
11     $h_s \leftarrow \text{lowerLvalue}(e_s)$ 
12     $p \leftarrow \text{newTemp}(); \text{emit}(p = \text{addr\_field } h_s, S.f)$ 
13    return  $p$ 
14  case pointer dereference  $*e_0$  do
15    return  $\text{lowerExpr}(e_0)$  (the r-value of  $e_0$  is already the pointer)

```

That covers expressions. Statements are different: a statement does not return a value handle, it just rearranges control flow and sometimes performs a side effect. The lowering for statements is where the basic-block structure is built up.

An `ExpressionStmt` lowers its inner expression (the handle returned is discarded). A `BlockStmt` lowers its children in order in the current block. A `ReturnStmt` lowers its expression (if any), then emits a `ret` terminator and closes the current block. A `BreakStmt` or `ContinueStmt` emits a `jmp` to the label stashed by the enclosing loop’s lowering and closes the current block. An `IfStmt` allocates fresh labels for the true-branch, the false-branch, and the join, emits a `br` on the lowered condition, lowers each arm into its respective new block ending with a `jmp` to the join, and continues in the join block.

The `ForStmt` is the one to spell out, because the anchor uses it and because it shows the four-block pattern that every loop inherits. The semantic tree node is `ForStmt(init, cond, step, body)` and the lowering produces four basic blocks (Figure 6.2). First we emit the `init` statement into the current block, then a `jmp` to a fresh `L_cond` label. `L_cond` lowers the condition expression and emits a `br` on its handle to either `L_body` or `L_exit`. `L_body` lowers the body statement (with the enclosing-loop labels stashed for any `break` or `continue` children) and ends with a `jmp` to `L_step`. `L_step` lowers the step expression and ends with a `jmp` back to `L_cond`, closing the loop. `L_exit` is where lowering of the surrounding code continues.

The full statement walker, including the `ForStmt` case just described, is collected in Algorithm 6.3. Reading the algorithm next to the prose above is the fastest way to see how the informal recipe for each statement kind translates into a uniform recursion that allocates fresh labels at the loop boundaries and threads the current-block handle through every branch.

Algorithm 6.3 is where the basic-block discipline of the IR actually gets enforced. Every `close()` closes a block with a single terminator and every `open()` starts a fresh block with no predecessors yet. The label stack $\mathcal{L}$ is the only piece of non-structural state the walker carries; it lets a deeply nested `break` resolve to the innermost loop’s exit label without an explicit ancestor pointer. The for-loop case (eight `Open/Close` pairs across four labels) is the most intricate; reading it next to Figure 6.2 below makes the four-block pattern visible at a glance, and the rest of the walker collapses to small variations on it.

The pattern generalises. `while` is the same shape minus the `init` and `step` blocks; `do-while` is the same shape with the body before the condition. Every loop is some specialisation of “`init`, `condition`, `body`, `step`, `exit`” wired together by terminators. Once you have written the `ForStmt` case once, every other loop in the language costs only a few lines.

The lowering walker, then, is a pair of mutually recursive functions over the typed tree: one for expressions (returning a value handle), one for statements (returning nothing). They share a state object that holds the current basic block, the slot table for the function, the fresh-temporary counter, the fresh-label counter, and the stack of enclosing-loop labels for `break` and `continue`. The two core walker classes are a few hundred lines of Java in `compiler-ir` and read like a textbook exercise — which is not an accident: the design intent is that they should.

Algorithm 6.3: Statement lowering with control flow. Loops follow the four-block pattern (init, condition, body, step), with the step block elided for while. Fresh labels are allocated at every loop and conditional boundary; break and continue resolve against the innermost enclosing loop via the label stack.

Input: a typed statement node s , the current block builder B , and a stack $\mathcal{L}$ of (continue-target, break-target) label pairs for enclosing loops

Output: the (possibly new) current block builder after s has been lowered

```

1  switch kind of  $s$  do
2    case ExpressionStmt of  $e$  do
3      | lowerExpr( $e$ )                                (value handle is discarded)
4    case BlockStmt of  $s_1, \dots, s_k$  do
5      | for  $i = 1$  to  $k$  do lowerStmt( $s_i$ )
6    case AssignStmt  $\ell = e$  do
7      |  $h \leftarrow \text{lowerExpr}(e)$ ;  $p \leftarrow \text{lowerLvalue}(\ell)$ 
8      | emit(store  $p$ ,  $h : \tau$ )                        ( $\tau$  is the type of  $\ell$ )
9    case ReturnStmt with optional  $e$  do
10     | if  $e$  is present then  $h \leftarrow \text{lowerExpr}(e)$ ; emit(ret  $h : \tau$ )
11     | else emit(ret)
12     | close( $B$ )
13   case IfStmt ( $c, s_t, s_f$ ) do
14     |  $L_t \leftarrow \text{newLabel}()$ ;  $L_f \leftarrow \text{newLabel}()$ ;  $L_j \leftarrow \text{newLabel}()$ 
15     |  $h \leftarrow \text{lowerExpr}(c)$ ; emit(br  $h$ ,  $L_t$ ,  $L_f$ ); close( $B$ )
16     | open( $L_t$ ); lowerStmt( $s_t$ ); emit(jmp  $L_j$ ); close( $L_t$ )
17     | open( $L_f$ ); lowerStmt( $s_f$ ); emit(jmp  $L_j$ ); close( $L_f$ )
18     | open( $L_j$ )                                     (remainder of the function continues here)
19   case WhileStmt ( $c, s_b$ ) do
20     |  $L_c \leftarrow \text{newLabel}()$ ;  $L_b \leftarrow \text{newLabel}()$ ;  $L_e \leftarrow \text{newLabel}()$ 
21     | emit(jmp  $L_c$ ); close( $B$ )
22     | open( $L_c$ );  $h \leftarrow \text{lowerExpr}(c)$ ; emit(br  $h$ ,  $L_b$ ,  $L_e$ ); close( $L_c$ )
23     | push( $(L_c, L_e)$ ,  $\mathcal{L}$ )
24     | open( $L_b$ ); lowerStmt( $s_b$ ); emit(jmp  $L_c$ ); close( $L_b$ )
25     | pop( $\mathcal{L}$ ); open( $L_e$ )
26   case ForStmt ( $s_i, c, s_p, s_b$ ) do
27     |  $L_c \leftarrow \text{newLabel}()$ ;  $L_b \leftarrow \text{newLabel}()$ ;  $L_p \leftarrow \text{newLabel}()$ ;  $L_e \leftarrow \text{newLabel}()$ 
28     | lowerStmt( $s_i$ ); emit(jmp  $L_c$ ); close( $B$ )
29     | open( $L_c$ );  $h \leftarrow \text{lowerExpr}(c)$ ; emit(br  $h$ ,  $L_b$ ,  $L_e$ ); close( $L_c$ )
30     | push( $(L_p, L_e)$ ,  $\mathcal{L}$ )
31     | open( $L_b$ ); lowerStmt( $s_b$ ); emit(jmp  $L_p$ ); close( $L_b$ )
32     | open( $L_p$ ); lowerStmt( $s_p$ ); emit(jmp  $L_c$ ); close( $L_p$ )
33     | pop( $\mathcal{L}$ ); open( $L_e$ )
34   case BreakStmt do
35     | ( $\_, L_e$ )  $\leftarrow \text{top}(\mathcal{L})$ 
36     | emit(jmp  $L_e$ ); close( $B$ )
37   case ContinueStmt do
38     | ( $L_c, \_$ )  $\leftarrow \text{top}(\mathcal{L})$ 
39     | emit(jmp  $L_c$ ); close( $B$ )

```

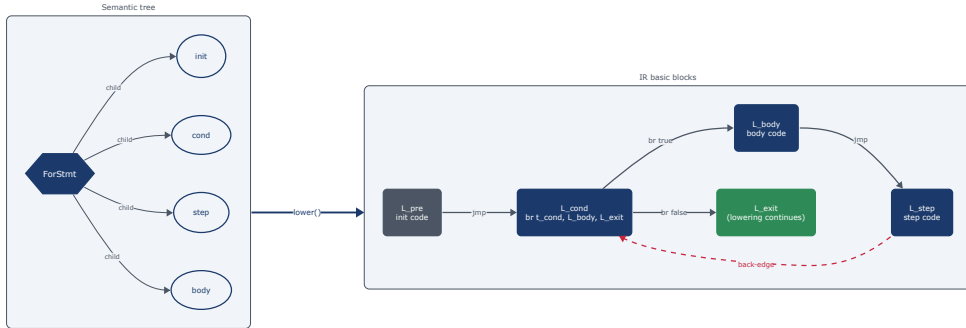


Figure 6.2: The typed tree’s `ForStmt` node fans out into four basic blocks. Each block ends in exactly one terminator; the loop’s back-edge is just a `jmp` from the step block back to the condition block.

The lowering produces IR. Whether that IR is well-formed — whether the optimiser and the back end can actually trust what it says — is a separate, sharper question, and it is the one we look at next.

6.5 Well-formedness as a written-down property

Suppose an optimisation pass, intending to hoist a load, accidentally copies a temporary name into a block that does not define it:

```
L0:
  t5 = add t3, #1:int32 : int32    ; t3 never defined
  jmp L1
```

The IR text is syntactically legal. It parses. Every line ends with a consistent type annotation. But when the back end asks “which register holds `t3`?” the answer is undefined, and whatever the back end invents will silently produce wrong output — a function that returns 4 when it should return 17, or worse, a function that returns whatever last touched `R1`. This is not a type error. The grammar of the IR accepts the fragment; the rules of three-address code do not catch it. It is, formally, a *well-formedness* violation: the IR is grammatically valid but structurally broken in a way the grammar cannot see.

The IR designer’s job, then, is twofold. The grammar in [Appendix D](#) pins down the shape of an instruction. A separate written contract, which we develop in this section, pins down the structural properties an entire function must satisfy *on top of* the grammar. Catching a violation of that contract at the IR level is much cheaper than catching it downstream as a spectacular failure during simulation, where the symptom (“returns the wrong number”) has no obvious connection to the cause (“a pass forgot to update a temporary name”).

There are eight invariants we ask the IR to satisfy, and they fall into three families. The first family is about *structural integrity* — the shape of control flow itself. The second is about *data integrity* — whether the values flowing through the instructions make sense. The third is about *frame integrity* — whether the function’s contract with the rest of the program is honoured. We take them in that order.

Structural integrity: the shape of control flow

Before we ask anything about the values inside a function, we have to agree on what the function’s control-flow graph even is. Two invariants together fix it.

Single terminator. Every basic block contains exactly one terminator instruction, and that terminator is the last instruction in the block. No block has zero terminators, because then control would run off the end into whatever happened to be the next block in memory, which is precisely the kind of undefined behaviour the IR was designed to forbid. No block has two, because then the second one would be unreachable. The invariant is trivial to check (scan each block for terminators, require exactly one, require it to be last) but its consequences are deep: the control-flow graph of a function is exactly the graph of blocks-to-blocks edges induced by the terminators, with no implicit edges. Compare this to the implicit-fall-through model of FRISC assembly itself, where every instruction silently transfers control to the next one in memory — in the IR we have made every transfer explicit, and it is the single-terminator invariant that lets us do so.

Reachable entry. The first block listed under `.blocks` is the entry block, and execution starts there unconditionally. The invariant requires that the entry block has no incoming edges from elsewhere in the function: nothing jumps to the entry. Otherwise the entry-block prologue — which sets up the stack frame in the back end — might run more than once on a single function call, which would corrupt the frame. The invariant is the IR’s structural way of saying “a function call enters the function exactly once.”

Data integrity: the meaning of values

With the shape of control flow fixed, three more invariants take care of what the instructions inside the blocks actually compute.

Definition before use. Every temporary mentioned as an operand must have been defined earlier in the function, on some path that actually reaches the use. The simple forward case — `t5 = add t2, t4` where `t2` and `t4` were both defined earlier in the same block — is easy. The interesting case is when the definition is in a predecessor block and the use is in a successor. The invariant is then phrased in terms of the control-flow graph: every use of a temporary must be *dominated* by a definition. Equivalently, on every path from the function entry to the use, there is at least one definition of the temporary before the use. The optimiser, when it propagates values across blocks in [Chapter 7](#), relies on this property; violating it would produce IR that the optimiser would silently miscompile. This is exactly the failure mode of the malformed fragment that opened the section.

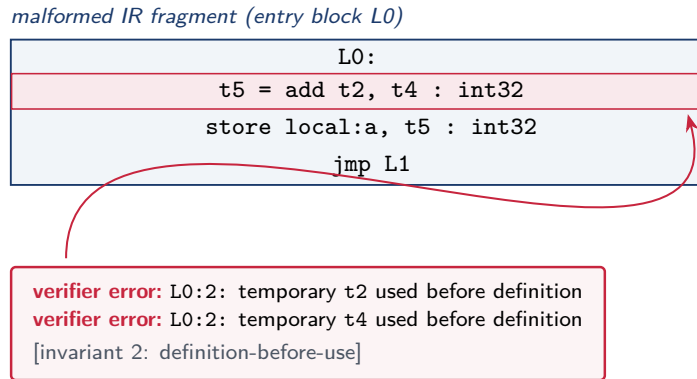


Figure 6.3: A malformed block: temporaries t2 and t4 are operands of the add but were never defined earlier in L0. The verifier rejects the IR with a precise diagnostic that names the offending block, the line, the temporary, and the invariant being violated. The front-end lowering walker cannot produce this fragment by construction — it allocates a fresh temporary before any use — but a careless optimiser pass that copies a temporary name across blocks can, which is what `--verify-each` is for.

Figure 6.3 shows what catching a violation looks like in practice. The diagnostic locates the failure to a single line and names the invariant, which is what we want from a checker: a verifier that says “something is wrong” without saying *what* would not save us any debugging time.

Type consistency. Every operation must accept operands of the types its opcode declares and produce a value of the type written after the colon. An `add` between an `int32` and a `float` is malformed; a `cmp_lt` that claims to return `int32` is malformed; a load of an `int32` from a pointer typed `ptr<bool>` is malformed. This is the IR’s analogue of the source-language type system from Chapter 5, and it is where the “typed” in “typed IR” earns its place in the name.

Unique labels. No two blocks in the same function share a label. The optimiser routinely picks blocks up, duplicates them, and moves them around, and the only way to do that without aliasing is to mint fresh labels each time. The lowering walker uses a monotonic counter; the optimiser does the same.

Frame integrity: the function’s contract

The last three invariants are about the boundary between a function and the rest of the program: the labels it advertises to its callers, the value it promises to return, and the slot table it commits to.

Target legality. Every `jmp` and every arm of a `br` names a label that exists in the same function. Cross-function branches are not legal; that is what `call` is for. This invariant lets the optimiser treat the control-flow graph as a closed object that does not need to track inter-procedural edges.

Return-type agreement. A function declared to return `int32` reaches a `ret t : int32`, never a bare `ret` and never a `ret t : float`. A function declared `void` reaches a bare `ret` on every path. Together these properties make function bodies composable: a caller that issued a `call f` can know, statically, the type of the returned value.

Slot-table coverage. Every `addr_of_symbol` in the function body names a slot listed in the `.slots` table; every slot listed in the table is reachable from at least one `addr_of_symbol` (or is a parameter slot, which is established by calling convention rather than by reference). The first half catches typos; the second half catches dead slots that the front end forgot to remove and that would otherwise waste frame space.

The IR well-formedness checklist. A function body is well-formed when, taken together:

- (1) every basic block has exactly one terminator, and it is the last instruction in the block;
- (2) every temporary use is dominated by at least one definition along every path from entry;
- (3) every opcode's operand and result types are consistent with the type catalogue;
- (4) every block label is unique within the function;
- (5) the entry block has no incoming edges;
- (6) every branch and jump target is a label in the same function;
- (7) every `ret` has the type declared in the function signature, and every `void` function reaches a bare `ret` on every path;
- (8) the slot table covers every `addr_of_symbol` target, and every listed slot is referenced.

A program is well-formed when every function in it is.

These invariants are checked by a single sweep called `IrVerifier` which lives in `compiler-ir` alongside the lowering. By default the verifier runs once, immediately after IR generation and before the optimiser sees the program: the lowering walker has to hand the back end something well-formed to begin with. There is also a `--verify-each` flag that re-runs the verifier after every optimisation pass, which is what we use when chasing down a bug: the offending pass announces itself by being the one that turned a well-formed program into a malformed one.

The eight invariants and the verifier between them give us a written contract for what the IR is, and what it forbids. They do not, by themselves, prove that the optimiser preserves semantics: an optimiser pass could turn a well-formed program into a different well-formed program with different behaviour, and the verifier would happily approve. That stronger property is the subject of [Appendix G](#), where each of the sixteen passes from [Chapter 7](#) comes with its own semantic-preservation argument. Well-formedness is the floor; semantic preservation is the ceiling; the optimiser sits between them.

With the notation defined and the invariants pinned down, the anchor is the next thing to read.

6.6 Worked example: lowering math_fibonacci_iter

The anchor program is about sixteen lines of C that computes the twentieth Fibonacci number iteratively, with the loop unrolled into the two-rolling-variables pattern. The source is Listing 1.

```

1 // EXPECT: 6765
2
3 int main(void) {
4     int n = 20;
5     int a = 0;
6     int b = 1;
7     int i;
8     int t;
9
10    for (i = 0; i < n; i++) {
11        t = a + b;
12        a = b;
13        b = t;
14    }
15
16    return a;
17 }
```

Listing 1: The anchor for chapters 3 through 6: an iterative Fibonacci that should return 6765 for $n = 20$.

Running it through `./run.sh --ir --00 program.c` produces the IR in Listing 2. We are looking at the IR before any optimisation, so the dump is exactly what the lowering walker emitted. It is verbose. We will walk it block by block.

The header is the program envelope we saw in Section 6.3. `main` returns `int32`, has a twenty-byte frame with four-byte alignment, and lists five slots: the loop bound `n` at offset 0, the rolling variables `a` and `b` at offsets 4 and 8, the loop counter `i` at offset 12, and the temporary `t` at offset 16. The order matches the order of declaration in the source because that is what the lowering walker does; nothing reorders the slots.

Block `L0` is the initialisation block. Each of the four initialisers in the source — `int n = 20`; `int a = 0`; `int b = 1`; and the `i = 0` from the for-loop’s init clause — becomes the same two-instruction pattern: take the address of the slot, store the constant into it. Four pairs of instructions, eight lines, no surprises. The block ends with `jmp L1`, transferring control to the loop’s condition test.

Block `L1` is the condition block. It reads `i` and `n` into fresh temporaries and compares them with `cmp_lt`, producing the Boolean result `t8`. The terminator is `br t8, L2, L4`: if `t8` is true, continue with the body; otherwise, exit the loop. Notice that we did not assume `i`’s value or `n`’s value were already in temporaries from the previous iteration — the lowering

```

1  .program
2
3  .func main():int32
4  .frame locals=20 bytes align=4
5  .slots
6      local n@0:int32
7      local a@4:int32
8      local b@8:int32
9      local i@12:int32
10     local t@16:int32
11  .blocks
12  L0:
13     t0 = addr_of_symbol local:n
14     store t0, #20:int32 : int32
15     t1 = addr_of_symbol local:a
16     store t1, #0:int32 : int32
17     t2 = addr_of_symbol local:b
18     store t2, #1:int32 : int32
19     t3 = addr_of_symbol local:i
20     store t3, #0:int32 : int32
21     jmp L1
22  L1:
23     t4 = addr_of_symbol local:i
24     t5 = load t4 : int32
25     t6 = addr_of_symbol local:n
26     t7 = load t6 : int32
27     t8 = cmp_lt t5, t7 : bool
28     br t8, L2, L4
29  L2:
30     t9 = addr_of_symbol local:a
31     t10 = load t9 : int32
32     t11 = addr_of_symbol local:b
33     t12 = load t11 : int32
34     t13 = add t10, t12 : int32
35
36     t14 = addr_of_symbol local:t
37     store t14, t13 : int32
38
39     t15 = load t11 : int32
40     store t9, t15 : int32
41     t16 = load t14 : int32
42     store t11, t16 : int32
43     jmp L3
44  L3:
45     t17 = addr_of_symbol local:i
46     t18 = load t17 : int32
47     t19 = add t18, #1:int32 : int32
48     store t17, t19 : int32
49     jmp L1
50  L4:
51     t20 = addr_of_symbol local:a
52     t21 = load t20 : int32
53     ret t21
54  .endfunc
55
56  .endprogram

```

Listing 2: Pre-optimisation IR for the Fibonacci anchor, emitted directly by the lowering walker.

walker reloads them each time through the loop, because at the IR level it has no idea what happened in L3. The optimiser will fix that.

Block L2 is the body. It implements three statements: $t = a + b$, $a = b$, $b = t$. The first reads a and b , adds them, and stores the result into t 's slot via $t14$ (the address of t 's slot) and $t13$ (the sum). The second reads b again — via $t11$ (the address it already held) and $t15$ (the reloaded value) — and stores it into a 's slot through $t9$ (the previously-computed address). The third reads t via $t14$ (the address) and $t16$ (the value), and stores it into b 's slot through $t11$. The block ends with `jmp L3`.

Take a moment to count the address-of-symbol and load instructions in L2. There are three address-of-symbol instructions and four loads. Two of the addresses ($t9$ and $t11$) reuse the addresses they computed earlier in the block; the load through $t11$ happens twice, once into $t12$ and once into $t15$. This is the kind of redundancy the optimiser will collapse: an unoptimised IR is honest about every memory operation the source semantically performs, while an optimised IR is honest about what actually needs to happen.

Block L3 is the step. It increments i by one — read i 's address, load its value, add the constant 1, store the result back — and jumps back to L1 to test the condition again. The four-instruction increment pattern is exactly what we predicted in [Section 6.4](#): address, load, add, store, in that order.

Block L4 is the exit. The loop is done; the function needs to return a . Read a 's address, load its value, return. Three instructions, including the terminator. Notice that we do not reuse $t9$ or $t10$ from the body — the lowering walker is functional: each temporary is fresh, and the optimiser is the one that gets to ask whether two reads of the same slot can share a temporary.

Twenty-two temporaries, five blocks, four loop iterations' worth of structure for a five-line C loop. The IR is doing exactly what we asked of it: making every memory read, every memory write, every arithmetic step, and every control transfer explicit. The cost is verbosity. The payoff is that the optimiser, looking at [Listing 2](#), can rewrite it without ever having to re-read the source.

[Chapter 7](#) will spend twenty pages on what that rewriting looks like, but it is worth getting a glimpse here. The optimised IR (file `listings/math_fibonacci_iter/intermediate_01.ir`) collapses L2 so that the two redundant address-of-symbol instructions and the duplicated load disappear; in the version we will see, the inner loop body is down to a handful of essential loads, two adds (the Fibonacci sum plus the loop-counter increment that the optimiser folded in), and three stores. The control-flow graph has actually shrunk: the optimiser folded the separate step block L3 into the loop body, so the loop now spans four blocks instead of five. The temporary count is unchanged in name (the optimiser preserves $t0$ through $t21$ where they survive) but more than a third of them are gone. That is the difference between IR that has been emitted and IR that has been polished, and [Chapter 7](#) is where the polish happens.

[Chapter 11](#) returns to this same IR file from another angle: it executes the file directly, instruction by instruction, against an abstract machine. Reading the lines of [Listing 2](#) as

instructions for that machine is the fastest way to convince yourself that the IR carries the entire meaning of the program.

Your turn

The companion repository at `frisccc-companion/ch06-ir/` is where you build a lowering walker of your own for the tinyC subset: integer-only locals, integer-only literals, the four binary arithmetic operators, comparisons, assignment, `if`, `while`, and `return`. The full case breakdown lives in the Project section below; here the only thing worth saying ahead of time is that the reference implementation is roughly two hundred lines of Java and the tests pretty-print your IR for diffing against a reference output, so the moment one of your cases is wrong you see exactly which line your output disagrees on. Budget an evening.

Practice

Exercise 6.1 (Applied · ★★★)

Lowering a nested expression by hand. Lower the expression $(x + 1) * (y - 2)$, where `x` and `y` are local `int` variables at slot offsets 0 and 4 respectively, applying the four-step pattern from [Section 6.4](#): (1) recursively lower the left sub-expression to a handle, (2) recursively lower the right sub-expression to a handle, (3) allocate a fresh temporary, (4) emit the instruction and return the temporary. Write out the full IR instruction sequence in the order the lowering walker emits it, annotating each instruction with the step that produced it. What handle does the top-level call return? How many temporaries did you allocate in total, and how many of them come from the `addr_of_symbol/load` pairs that each variable reading requires?

Exercise 6.2 (Applied · ★★★)

Tracing a while-loop lowering. Consider the following tinyC fragment:

```
int i = 0;
int s = 0;
while (i < 3) {
    s = s + i;
    i = i + 1;
}
return s;
```

Apply the lowering rules from [Section 6.4](#) step by step. Name the basic blocks `L0` through `L3` in the order the lowering walker allocates them (entry, condition, body, exit). For each block, list its instructions and terminator. Identify which terminator connects the body back to the condition, and which terminator leaves the loop. How

many blocks does the walker produce? How many temporaries?

Exercise 6.3 (Applied · ★★☆☆)

Spot the well-formedness violation. Each of the following IR fragments violates exactly one invariant from the checklist in [Section 6.5](#). Identify the violated invariant by number and explain in one sentence why it is broken.

- (a) A block that contains the sequence `ret t2` followed by `store t0, #1:int32 : int32`. Which invariant does this break, and what does the broken invariant say?
- (b) An instruction `t5 = add t3, #1:float : int32` where `t3` has type `int32`. Which invariant governs operand and result types?
- (c) A function whose only block is labelled `L0` and ends with `jmp L0`. Which invariant do you need to check, and does this fragment actually violate it? (Hint: read invariant 5 carefully.)
- (d) A function that declares `local w@8:int32` in its slot table but contains no `addr_of_symbol local:w` anywhere in its body. Which invariant covers this case, and what is the motivation given in [Section 6.5](#) for requiring it?

Exercise 6.4 (Applied · ★★☆☆)

Three-address verbosity audit. Open [Listing 2](#) (the pre-optimisation IR for the anchor program). Count every temporary in the listing: first the total, then those whose defining instruction is `addr_of_symbol`, then those whose defining instruction is `load`. If FRISCCc had been redesigned so that a slot name could appear directly as an operand — so `t = add local:a, local:b : int32` were legal IR — how many temporaries would the same function require? Argue in two or three sentences why the compiler’s authors chose the verbose form, using the discussion in [Section 6.1](#) as evidence.

Exercise 6.5 (Applied · ★★★)

Tracing the slot table. The slot table of the anchor’s `.func main():int32` lists five slots: `n@0`, `a@4`, `b@8`, `i@12`, `t@16`. Without running the compiler, answer the following. First: what is the total frame size in bytes, and which `.frame` directive line in [Listing 2](#) records it? Second: which slot would occupy offset 20 if the source declared a sixth `int` local `u` at the end of the variable list? Third: suppose the optimiser eliminates all uses of `t` and removes it from the slot table, renumbering offsets to keep them contiguous. Write the updated `.slots` block, and state which well-formedness invariant from [Section 6.5](#) the original listing would violate if `t` remained in the slot table but was never referenced by any `addr_of_symbol`.

Project

Exercise 6.6 (Lab · ★★★)

Project: a tinyC IR generator. Complete the IR lowering walker for the tinyC subset in the companion repository.

Open `IrLowerer.java` in the starter’s `com/frisccc/tinyc/ir/` package. The data model (`IrType`, `IrValue`, `IrInstr`, `IrTerminator`, `IrBlock`, `IrFunction`, `IrProgram`), the mutable builder (`IrBuilder`), the pretty-printer (`IrPrettyPrinter`), and the AST node hierarchy (`ast/Expr.java`, `ast/Stmt.java`) are already in place. The already-implemented cases — literal constants, identifier loads, assignment, function calls, and return — show exactly what the four-step pattern looks like. Your job is the four `// TODO: implement` markers:

TODO 1 `Expr.BinOp` — lower `+`, `-`, `*`, `/`, `%` to the corresponding IR opcodes (`add`, `sub`, `mul`, `div`, `mod`).

TODO 2 `Expr.Cmp` — lower `<`, `<=`, `>`, `>=`, `==`, `!=` to `cmp_lt`, `cmp_le`, `cmp_gt`, `cmp_ge`, `cmp_eq`, `cmp_ne`.

TODO 3 `Stmt.If` — allocate labels for the true-arm, the false-arm, and the join; lower the condition; close the current block with `br`; lower each arm in its own block; open the join block and continue.

TODO 4 `Stmt.While` — allocate labels for the condition, body, and exit blocks; emit the four-block pattern from [Section 6.4](#).

When all four are done, run

```
mvn test -pl ch06-ir
```

from the companion root. All thirteen diff-based tests must pass. Each test constructs a tinyC AST, lowers it, pretty-prints the result, and compares it character-for-character against a reference `.ir` file in `ch06-ir/tests/src/test/resources/ch06/`. The reference solution is roughly eighty lines of lowering logic. Budget an evening.

Stretch goal: extend the lowerer to handle short-circuit `&&` and `||` via control-flow lowering rather than a single IR instruction. The technique is the same one FRISCCc’s `LogicalExpressionGenerator` uses: lower `a && b` as an if-shaped graph that skips evaluating `b` when `a` is already false.

Notes and further reading

Three-address code is one of the oldest ideas in compilation; Aho, Sethi and Ullman codify it in the Dragon Book [2] and Cooper and Torczon spend most of *Engineering a Compiler* [18] working in it as a uniform medium for optimisation. Appel’s *Modern Compiler Implementation in ML* [6] is the classic short introduction to IR-shaped thinking, and Muchnick’s *Advanced Compiler Design and Implementation* [62] remains the encyclopaedic reference for everything an IR can do once it exists. Wirth’s *Compiler Construction* [85] is the dissenting opinion worth reading: he argues, persuasively, that for many compilers

an explicit IR is more ceremony than it is worth, and emits machine code directly from a recursive-descent walker. We disagree, for the reasons sketched in [Section 6.1](#), but the case is better than its reputation.

Static single assignment originates in the work of Cytron, Ferrante, Rosen, Wegman, and Zadeck [21], and its modern manifestation in production compilers is best read about in the LLVM paper [53]. We have already explained why FRISCcc does not use SSA; the reader curious to try is encouraged to attempt a Practice-tier SSA-construction project on top of the `ch06-ir/solution/` reference, where the unique-name discipline FRISCcc already enforces means you only have to add ϕ -functions.

The semantic-preservation arguments that justify each optimisation pass living downstream of the IR — which we have here promised rather than delivered — live in [Appendix G](#), one section per pass.

Part III

The Back End

Chapter 7

Optimizing without breaking things

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”
— Donald Knuth

At the end of [Chapter 6](#) we held a typed intermediate representation: a flat sequence of three-address instructions, every value annotated, every basic block ending in exactly one terminator, the whole thing verified well-formed. It was correct. It was also, honestly, embarrassing. Reading the IR for our chapter anchor, `real_prime_sieve`, we could see the lowering walker emitting the address of `p` four times in the same block, loading the same slot two lines in a row, computing `p * p` once for the loop condition and again to seed the inner loop, and threading control through a block whose only purpose was to issue a single `jmp` to its neighbour. None of that was a bug. The lowering walker is deliberately literal: it makes every memory read, every memory write, and every control transfer explicit, and it does so without looking sideways at what the surrounding code is up to. The price of literal lowering is verbose IR. The price of unfixed verbose IR is the back end paying for every redundancy in cycles and bytes.

That gap, between honest verbose IR and the lean code it ought to become, is where this chapter does its work. We have a uniform medium, the IR, and a small library of *passes*: programs that take an IR program and produce a new IR program that, by some local argument, has the same observable behaviour and is in some local sense better. Sixteen passes, each modest, each addressing one species of waste — redundant arithmetic, redundant copies, dead stores, unreachable blocks, loop-invariant computations, calls to trivial functions, comparisons whose outcome we already know. None of them is dramatic on its own. Stacked, and iterated to a fixed point, they turn the lowering walker’s verbose first draft into IR that looks like something a careful programmer might have written by hand.

We should be clear about what this chapter is *not*. It is not about chasing the last few percent on a benchmark suite — FRISCCc is a teaching compiler, not a production compiler, and the passes we are about to meet are a small but representative collection rather than the encyclopaedic catalogue of Muchnick’s textbook [62] or the production-grade machinery of LLVM [53]. It is not about peephole optimisation on FRISC assembly; that work lives downstream, in [Chapter 8](#), where we lower to instructions and can finally look at addressing modes and register allocation. And it is emphatically not about formal proofs. Each

pass we describe has a semantic-preservation theorem behind it, but the theorems live in [Appendix G](#), one section per pass, and the narrative here stays in the imperative mood: *what* we do, *why* the redundancy is safe to eliminate, and *when* the pipeline asks that pass to run. The proofs are the floor under the floor; we will walk on top of it.

Knuth’s quote is more often misused than understood. He was warning against micro-optimising code whose hot path you have not measured, not licensing programmers to write sluggish code on principle. For us, sitting between a literal lowering walker and a FRISC code generator, the situation is sharper than either reading. We are not guessing about the hot path; we know, by construction, that the lowering emits every memory access the source semantically performs, and we know that the back end has no way to recover from that. The optimisations we will discuss are not premature; they are the only ones positioned to fire at all.

7.1 What we mean by “optimization”

Take a small fragment of IR. The source program contains `int x = 2 + 3 * 4;` and the lowering walker, reading the expression bottom-up, emits

```
1   t0 = addr_of_symbol local:x
2   t1 = mul #3:int32, #4:int32 : int32
3   t2 = add #2:int32, t1 : int32
4   store t0, t2 : int32
```

which is correct, and is also doing more work than `x = 14`. Three constants are knowable at compile time; the IR temporaries `t1` and `t2` are computing values that the optimiser, with nothing more than a calculator and a willingness to look at the operand types, can compute itself. After it does:

```
1   t0 = addr_of_symbol local:x
2   store t0, #14:int32 : int32
```

Two instructions instead of four. Two temporaries gone. One multiplication and one addition simply do not happen at run time. The transformation is local — the optimiser looked at one block and never crossed a label — and it is correct, because integer arithmetic in the IR has the same two’s-complement wraparound semantics as integer arithmetic on FRISC, so `2 + 3 * 4` on paper agrees with `2 + 3 * 4` on the hardware. We will say more about that in [Section 7.2](#).

That little rewrite is what we will call an *IR-to-IR pass*: a function that takes a well-formed IR program and returns a well-formed IR program with the same observable behaviour. Two things in that definition deserve attention. The first is *well-formed in, well-formed out*: every pass is required to respect the eight invariants from [Chapter 6](#), and the validator

(`IrOptimizationValidator`, which calls the verifier you met in the previous chapter) is run before and after the pipeline to enforce it. A pass that turns a well-formed IR into a malformed one is a bug, caught early. The second is *same observable behaviour*: the output of running the program — its return value, its print output, its memory traffic to externally visible locations — must not change. Internal details may. Temporaries can disappear; blocks can disappear; even slots can disappear if no surviving instruction references them. What we owe the user is that the program prints what it printed before and returns what it returned before.

The word “optimisation” is mildly misleading here. We are not guaranteeing that the rewritten program is faster, or smaller, or better by any single metric. We are guaranteeing that the rewritten program is the same program written more economically by some local notion of economy. Most of the time the result is faster and smaller; occasionally a transformation pays in code size to save on cycles, or vice versa. The distinction the literature draws between *transformation* (any IR-to-IR rewrite that preserves semantics) and *optimisation* (a transformation that improves a chosen metric) is one we will gloss over in the prose, because every pass we ship is both, but it is worth knowing the words exist [62].

Key insight

A pass is *semantics-preserving* if, for every well-formed input program P that satisfies the IR invariants, the output program $\text{pass}(P)$ is well-formed and produces the same observable behaviour as P on every input. The full statement, including the definition of observable behaviour for FRISCCc’s IR, is the subject of [Appendix G](#). The chapter narrative discusses each pass’s correctness intuitively; the appendix discharges the obligation formally, one pass at a time.

A pass is, mechanically, a Java class implementing the `IrPass` interface — one `name()` method, one `run(program, context)` method that returns a `PassResult` carrying the new program and a changed flag. Sixteen such classes live in `compiler-opt/src/main/java/.../passes/`, organised into a handful of thematic subpackages: `arith`, `cast`, `shift`, `controlflow`, `flow`, `temps`, `memory`, `loop`, `range`, `inline`. Each is small (a few hundred lines), each focuses on one species of redundancy, and each is composable with the others through a fixed-point loop we will spell out in [Section 7.7](#).

The rest of this chapter walks the sixteen passes in five thematic groupings. We start with constants and arithmetic, because they are the simplest to explain and the most visibly satisfying. We then move to control-flow tidying, where the optimiser learns to prune blocks and edges; to the data-flow family, which is the meat of the pipeline and the source of most of the savings; to loops, which deserve their own attention because the inner loop is where any real program spends its time; and finally to cleanup, where a small inliner and a value-range simplifier polish whatever remains. With those five groupings in hand, we will be ready to look at the pipeline as a whole and at the question of why the passes are ordered the way they are.

7.2 Constants and arithmetic

The cheapest savings come from computations whose operands are known at compile time, and the loop bound in `real_prime_sieve` is a perfect example. The source declares `p * p <= 200` as the outer loop’s exit condition, and we will eventually want to reason about how `p * p` flows through the loop. Set that aside for a moment; consider instead the inner-loop initialisation, `int j = p * p;`, where `p` happens to be 2 the first time control reaches that line. The lowering walker does not know that: it emits a generic multiply against the value loaded from `p`’s slot. But once an earlier pass has propagated `p`’s known value to the multiply (we will see how, in [Section 7.4](#)), the multiply becomes `mul #2:int32, #2:int32 : int32`, and at that point the optimiser can do the arithmetic itself and replace the instruction with the constant `#4:int32`. The pass that performs that replacement is *typed constant folding*.

`TypedConstantFoldingPass` walks each block looking for instructions whose operands are all literal constants of the right types and whose opcode is one that the optimiser knows how to evaluate at compile time — the four binary arithmetic operators and the remainder (add, sub, mul, div, mod), the bitwise operators (and, or, xor, shl, shr), the six comparisons, the unary operators (neg, not), and the casts. When it finds one, it computes the result, replaces the instruction with the literal value bound to the destination temporary, and lets a later pass (`CopyPropagationPass`) carry the constant forward into uses. Concretely, given

```

1   t1 = mul #2:int32, #2:int32 : int32
2   t2 = add t1, #1:int32 : int32

```

folding produces `t1 = #4:int32`; the addition is not foldable on the same scan because `t1` is a temporary, not a constant, but on the very next iteration of the pipeline, after copy propagation has rewritten `t1` into the literal `#4:int32` at the use site, folding fires again and produces `t2 = #5:int32`. The interaction is the first hint that the pipeline needs an outer fixed-point loop, and we will return to it.

The recipe the pass follows is short enough to write down in full ([Algorithm 7.1](#)): a single forward sweep over every block, testing each instruction against the same two-part condition (all operands literal, opcode in the foldable set), evaluating when the test passes, and rewriting in place. There is no fixed-point loop inside the pass and no cross-block bookkeeping; the iteration story lives one level up, in the pipeline driver.

[Algorithm 7.1](#) makes two design choices visible that are easy to miss in prose. The first is that folding is utterly local — no operand is consulted beyond the instruction being examined, and no block is consulted beyond the one being walked — which is why the correctness argument that follows reduces to a per-opcode table of evaluation rules rather than a global theorem. The second is that the pass reports its `changed` flag at the function level, not the instruction level: the pipeline driver of [Section 7.7](#) only needs to know whether the IR moved during the sweep, not which instructions moved. The simplicity is the point. Folding is the smallest pass in the optimiser, and the algorithm box exists to make that smallness concrete.

Algorithm 7.1: Typed constant folding: replace instructions whose operands are all literal constants with a single literal binding.

Input: An IR program P with well-formed blocks.

Output: An IR program P' in which every instruction whose operands are all literal constants has been replaced by a constant-bound copy.

```

1 changed  $\leftarrow$  false
2 for each function  $f \in P$  do
3   for each block  $B \in f$  do
4     for each instruction  $I \in B$  in order do
5       if opcode( $I$ ) is foldable and every operand of  $I$  is a literal then
6          $v \leftarrow$  evaluate  $I$  on its literal operands using the IR's typed semantics
7         replace  $I$  by dest( $I$ )  $\leftarrow v$ 
8         changed  $\leftarrow$  true
9 return ( $P$ , changed)

```

The correctness argument is the one we hinted at in the opening: integer arithmetic in the IR is defined to behave the same way it does on FRISC, which is the same way it does on every two's-complement 32-bit machine. $2 + 3 * 4$ evaluated by Java's long-then-cast-to-int in the optimiser produces the same two's-complement bit pattern the FRISC simulator computes for $2 + 3 * 4$ (the calling convention for the software multiply helper `F_MUL` is introduced properly in [Chapter 8](#)). The same goes for the bitwise operators, the comparisons, the unary operators, and the integer casts. The fixed-point arithmetic on `fixed16_16` needs a little more care — multiplication is not the bitwise operation on the raw 32-bit pattern, it is a shift after a 64-bit intermediate — and the helper class `Q16FloatSemantics` in the same package encodes the rule. The full preservation argument lives in [Appendix G](#).

What folding does not do. Typed constant folding only fires when *all* operands of an instruction are syntactically literal constants. The instruction `t1 = add t0, #1:int32 : int32`, where `t0` is a temporary whose value happens to be known elsewhere in the function, is not foldable by this pass alone — the pass does not look across instructions to discover that `t0` was bound to a literal. That work is the job of `GlobalValuePropagationPass` in [Section 7.4](#), which propagates known constants into the operand positions where folding can then fire. The split is deliberate: folding is a purely local rewrite and is correspondingly trivial to prove correct, while propagation is a dataflow problem and is correspondingly more delicate. Keeping them separate keeps each pass small.

Three further arithmetic passes round out the family. The first is `Int32ArithmeticPass`, which knows the algebraic identities that fall out of two's-complement integer arithmetic without needing either operand to be a literal. `t = add t0, #0:int32` collapses to `t = t0`; `t`

`= mul t0, #1:int32` collapses to the same; `t = mul t0, #0:int32` collapses to `t = #0:int32`; `t = sub t0, t0` collapses to `t = #0:int32`. None of these requires either operand to be known; they are theorems about the arithmetic, and they are sound because the IR's `int32` arithmetic is the arithmetic of $\mathbb{Z}/2^{32}\mathbb{Z}$ with signed interpretation. The same identities for the bitwise operators (`xor t, t` is zero; and `t, #0` is zero; or `t, #-1` is all ones) live in the same pass.

The second is `Int32ShiftPass`, which specialises the same idea for power-of-two multiplications and divisions: `mul t, #256:int32` becomes `shl t, #8:int32`, and the symmetric case for division becomes an arithmetic right shift. The win is real on FRISC, where the shift instructions take a single cycle and the software-implemented multiply (`F_MUL`, which we will meet properly in [Chapter 8](#)) costs upwards of thirty. Reducing even one multiply per iteration in a hot loop changes the run time visibly. The pass is careful about signed division, where shifting is not the same as dividing when the dividend is negative, and it refuses to rewrite in that case — we are not in a position to miscompile in pursuit of a single shift.

The third is `CastSimplificationPass`, which eliminates casts that the lowering walker emitted defensively but which carry no information. The clearest case is `trunc` followed by `sext` to the same width, which is the identity on sign-extended values, and the symmetric case is `sext` followed by `trunc`. Two casts in, zero casts out. Likewise, an `int32` cast to `int32`, or a `ptr<int32>` `ptrcast` to `ptr<int32>`, is the identity and the pass deletes it. The opportunities are not glamorous; they accumulate because the lowering walker, being uniform, sometimes emits cast pairs when the source already had the right type. Cleaning them up before the back end sees them avoids generating instructions that would do nothing on FRISC anyway.

These four passes — folding, arithmetic identities, shift strength reduction, and cast simplification — share a common shape. Each one is a single forward sweep through every basic block of every function; each one recognises a small number of patterns; each one either replaces an instruction with a cheaper one or leaves it alone. The fact that two of them (folding and arithmetic identities) can each enable the other is the source of the iteration story — after folding produces a new constant, an arithmetic identity may now apply; after an arithmetic identity reduces an expression, folding may now apply to its consumer. The pipeline runs the whole sequence until a full sweep produces no changes; we will look at why that terminates in [Section 7.7](#).

7.3 Tidying the control-flow graph

The lowering walker, by virtue of being structurally recursive, sometimes produces basic blocks whose only job is to issue a `jmp` to the next block. The clearest case in `real_prime_sieve` is the early exit from the inner sieve loop: the source contains a `while` whose body sometimes finishes via a `break`-shaped path and sometimes falls off the bottom, and the lowering walker, treating each case uniformly, produces a join block whose only instruction is `jmp L_step`. That block exists; it is well-formed; it is also a wart. Walking through it at run time costs a

taken branch, and at codegen time it costs an unconditional jump. Both are cheap. Neither needs to exist. The pass that removes such warts is `ControlFlowSimplificationPass`.

Key insight

A common pattern across the optimiser is to split a transformation into two passes: one recognises the redundancy and rewrites references, the other physically removes the now-orphaned construct. Keeping recognition and deletion separate keeps each pass small, gives the validator a chance to check well-formedness in between, and mirrors how a road crew closes a redundant junction — traffic gets rerouted first, the asphalt comes up later.

The pass does two related things. First, it rewrites branches on constant Booleans: a `br #true:bool, L_then, L_else` becomes `jmp L_then`, and the `L_else` side is no longer reachable from this edge. Second, it collapses jump-only blocks: if block `L_mid` contains nothing but the terminator `jmp L_target`, then every predecessor of `L_mid` can have its branch retargeted to `L_target`, and `L_mid` becomes garbage. The pass does the retargeting and leaves the dead block in place; a follow-up pass, `UnreachableBlockEliminationPass`, removes it — the two-step rhythm the keyinsight above describes, in its first concrete instance.

Consider a loop, abridged to its control skeleton. Before simplification, the IR contains seven blocks — an entry, a condition test, the body, a jump-only block that does nothing but `jmp` to its successor, a join, an unreachable block the lowering walker left behind, and the return — wired as in [Figure 7.1](#). The jump-only block does nothing useful; the simplification pass retargets its predecessor’s terminator directly at its successor, and the block becomes orphaned.

[Figure 7.1](#) is the keyinsight’s two-step rhythm in miniature: the left-hand graph contains the dashed `L3` that the simplifier recognises as redundant, the right-hand graph contains the cleaner topology after its predecessor edges have been retargeted, and the dashed node disappears entirely once the unreachable-block pass runs. What the picture lets us see, and what the prose can only describe, is that the rewiring is mechanical — predecessors of `L3` change their terminator’s target, nothing in the body of any other block has to move — which is precisely why splitting recognition from deletion costs nothing in expressive power. The same shape will appear again every time we encounter a “recognise then remove” pair, and we will not belabour it.

`UnreachableBlockEliminationPass` is the obvious follow-up. It computes the set of blocks reachable from the entry by following terminator edges, and it deletes every block not in that set. “Reachable” here is purely syntactic — the pass does not try to reason about runtime values, so it will not delete a block that is guarded by a comparison that always evaluates false; that is the job of `ValueRangeSimplificationPass` downstream. The pass is a graph traversal followed by a list filter; it is forty lines of genuine work plus the usual bookkeeping. Together with the simplifier it removes the join block we just orphaned, the `L_else` block that became dead when a constant-true branch turned into a `jmp`, and the after-return tail that the lowering walker occasionally emits when a `return` sits in the middle of a block (it shouldn’t, but if a later pass introduces one it might).

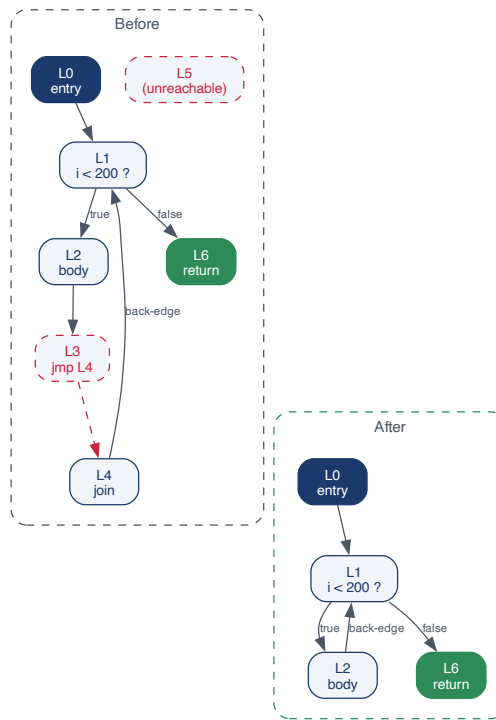


Figure 7.1: The control-flow graph of a loop, before and after `ControlFlowSimplificationPass` followed by `UnreachableBlockEliminationPass`. The dashed jump-only block (L3) does nothing but `jmp L4`; its predecessor edges are redirected to its single successor by the simplifier, then the block — along with the unreachable L5 — is deleted by the unreachable-block pass. Seven blocks become four; one taken branch per iteration is saved.

The correctness story for both passes is brief enough to state in prose. Redirecting a branch from `L_mid` to `L_target`, where `L_mid` contains only `jmp L_target`, is sound because in the original program any path through `L_mid` immediately leaves it; the only effect of `L_mid`'s body on the surrounding semantics is that the program counter visits it, and the IR's observable behaviour does not include the program counter. Deleting an unreachable block is sound because by definition no execution of the program ever runs its instructions; the deletion changes the static text of the IR but not its operational trace. Both arguments are made precise in [Appendix G](#).

The two control-flow passes do not save dramatic numbers of cycles on their own — a few taken branches per function, perhaps — but they buy something else that is more valuable: a cleaner control-flow graph for the downstream passes to reason over. Loop detection, dataflow analysis, and dominance computation all benefit from a graph that does not contain trivially redundant blocks, and the more often the optimiser can simplify the graph in place, the more often the analyses downstream see the structure they really want. Onward, then, to where the cycles actually live.

7.4 Following values around

The lowering walker, having emitted `store t0, #200:int32 : int32` into the loop-bound slot in `L0`, does not later emit `#200:int32` where the bound is read. It emits `addr_of_symbol local:n; load t, addr; ...` and then uses `t` downstream. The constant `#200` is buried in the slot's store and never re-emerges — even though, by any honest reading of the program, the value of `n` on every reachable load is precisely `#200`. Multiply that pattern by the dozens of slots a real function declares, and a substantial fraction of every block's loads are reading constants that the optimiser, with a little bookkeeping, could surface directly.

That is the kind of question the next family of passes is built to answer: *what value is this temporary, or this slot, holding at this point in the program?* The lowering walker, by being uniform, treats every load and every store as opaque. The optimiser, by following values around, learns which loads are reloading something already in a register, which stores write to slots that are immediately overwritten, and which temporaries are aliases for other temporaries. Sixteen lines of IR shrink to ten; ten shrink to seven. The savings stack.

Six passes are in this family, and we will tell them as a single story in the order the pipeline runs them. The order matters because each pass exposes opportunities for the next, and the family as a whole runs to a fixed point. We will look at each pass on its own and then look at the dependency between them.

Global value propagation: constants travel

Returning to the buried `#200` of the opening example, `GlobalValuePropagationPass` is the pass that re-emerges it.

The pass performs a conservative dataflow analysis over the control-flow graph. For each (slot, block) pair, it asks: is there a single constant that the slot definitely holds on entry to this block, regardless of which predecessor we came in from? If yes, every load of that slot in that block is rewritten to use the constant. The analysis is a meet-over-paths in the lattice of “constant or unknown”, where the meet of two distinct constants is unknown; it converges to a fixed point, and FRISCCc’s implementation realises the Kildall-style data-flow framework [44] — folklore in the field since the early 1970s — as a round-robin iterative solver that re-scans every block until the in/out states stop changing. A constant in a slot becomes a constant on every use the optimiser can reach, and what was a load is now a literal — which is exactly the diet copy propagation has been waiting for.

The Kildall framework is general enough to be worth seeing once in the abstract, because it underlies not only global value propagation but also the dataflow components of dead-store elimination, load forwarding, and value-range simplification we will meet later. [Algorithm 7.2](#) states the framework’s forward variant as a worklist algorithm driven to a fixed point. FRISCCc realises the same fixpoint as a round-robin iterative solver — re-scanning every block each round until nothing changes — which is one valid scheduling of the same dataflow equations; the worklist version is shown here because it makes the termination argument explicit.

[Algorithm 7.2](#) is the engine. Plugging *constant-or-unknown* in as the lattice and “what does this block do to each slot’s known value” as the transfer function gives global value propagation; plugging *set of definitely-live slots* in gives liveness; plugging *interval of possible values* in gives the analysis that drives `ValueRangeSimplificationPass` later in the chapter. The reason the framework deserves a box of its own is that termination and correctness depend on properties of the inputs — the lattice must have finite height, the transfer functions must be monotone — and those properties are easier to check once for each lattice than to re-prove inside each pass. The worklist version states the algorithm in a form general enough for the proof in [Appendix G](#) to discharge it once.

Copy propagation: temporaries are aliases

The next pass in line is `CopyPropagationPass`, and it addresses a different sort of redundancy. The lowering walker, when it lowers an assignment `a = b`, emits a load of `b`’s slot into a fresh temporary and then a store from that temporary into `a`’s slot. Subsequent reads of `a` go through a fresh `addr_of_symbol local:a; load` pair, ignoring the fact that we still have `b`’s value in scope. Worse, the algebraic simplifier in [Section 7.2](#) produces instructions of the shape `t5 = t4` when it collapses an arithmetic identity, and those copies, like all copies, want propagating.

Within a single basic block, the pass maintains a map from each temporary to its currently-known source: when it sees `t5 = t4` (or `t5 = #3:int32`, since the constant case is handled uniformly), it records `t5 ↦ t4` and then rewrites every subsequent use of `t5` in the block to reference `t4` directly. The copy instruction itself is left in place — a later pass,

Algorithm 7.2: Generic worklist algorithm for forward dataflow analysis over the control-flow graph, in the Kildall framework.

Input: A control-flow graph $G = (B, E)$ with entry block b_0 ; a join-semilattice $(L, \sqcup)$ of abstract values with bottom element $\perp$ and top element $\top$; an initial abstract value $init \in L$ for the entry; a monotone transfer function $trans_b : L \rightarrow L$ for each block b .

Output: A map $out : B \rightarrow L$ such that, for every b , $out[b]$ is a fixed point of the dataflow equations.

```

1 for each  $b \in B$  do
2    $\sqcup$   $in[b] \leftarrow \top$ ;  $out[b] \leftarrow \top$ 
3  $in[b_0] \leftarrow init$ 
4  $W \leftarrow \{b_0\}$  (worklist of pending blocks)
5 while  $W \neq \emptyset$  do
6   pick and remove some  $b$  from  $W$ 
7    $v \leftarrow trans_b(in[b])$ 
8   if  $v \neq out[b]$  then
9      $out[b] \leftarrow v$ 
10    for each successor  $s$  of  $b$  in  $E$  do
11       $u \leftarrow \sqcup \{out[p] : p \text{ predecessor of } s\}$ 
12      if  $u \neq in[s]$  then
13         $in[s] \leftarrow u$ 
14        add  $s$  to  $W$ 
15 return  $out$ 

```

`DeadTempEliminationPass`, will notice that `t5`'s only definition has no live uses and remove it. Concretely, before:

```
1   t4 = load t3 : int32
2   t5 = t4
3   t6 = add t5, #1:int32 : int32
```

After copy propagation (but before dead-temp elimination):

```
1   t4 = load t3 : int32
2   t5 = t4
3   t6 = add t4, #1:int32 : int32
```

The pass deliberately operates within a single block. Cross-block copy propagation would need either SSA or the same dataflow machinery as global value propagation, and `FRISCCc`'s implementation keeps things local — the pass is forty lines of work, and the opportunities it cannot catch within a block are typically caught on the next iteration of the pipeline after a different pass has moved them into the same block.

Dead-temp elimination: the unused disappear

After copy propagation rewrites `t6 = add t5, #1:int32` into `t6 = add t4, #1:int32`, the instruction `t5 = t4` is still there — but nothing in the block reads `t5` any more. It is a definition with no uses: dead weight. The pass that notices this and removes it is `DeadTempEliminationPass`. For each temporary in each function it collects the set of instructions that mention it as an operand; if that set is empty and the defining instruction has no side effects, the instruction is gone.

“Side effects” is the qualification that matters. A load of a slot has no side effects — it can be removed if its result is unused. A store writes to memory and is not removable just because no immediate successor reads it (that's a different pass; we will get there in two paragraphs). A call may print, may read, may modify global state; the pass treats every call as having side effects and leaves it alone. The arithmetic, address-of, and cast opcodes are pure: their result is the only thing they produce, and if no one reads it the instruction is free to vanish.

Applied to the copy-propagation example above, the dead-temp pass sees that `t5`'s only definition is `t5 = t4`, and that nothing in the block (or anywhere else in the function) reads `t5` now that the copy propagation has rewritten its uses. The instruction is deleted. One temporary fewer; one allocation slot the back end will not have to assign a register to — but more importantly, with the dead temporaries gone, the stores those temporaries used to feed become visible candidates for the next pass in line. Where dead-temp elimination tidies the temporary graph, dead-slot-store elimination does the equivalent for memory.

Dead-slot-store elimination: stores with no readers

Stores are heavier hammers than temporaries, and they need their own pass. Consider the body of the inner sieve loop, where the lowering walker emits a store into j 's slot at the end of each iteration and a load from the same slot at the top of the next. After load forwarding (next subsection) has rewritten the load, the store is still there — but if the only thing that reads j in the next iteration is now reading through a different temporary, the store may have no consumers at all on the path it sits on. If the next thing that happens to j 's slot is another store, without any intervening read, the first store is dead.

`DeadSlotStoreEliminationPass` tracks, for each slot, the sequence of stores and loads that touch it within a function. A store is dead if the slot is overwritten before any load reaches it and is not read after the function returns (i.e. is not an externally-visible memory location). For function-local slots the second clause is automatic. For slots backing parameters or globals the pass is conservative: it leaves stores to externally visible locations alone.

The correctness argument relies on a subtle point about memory in the IR: every load and store carries the address it operates on, and the optimiser only declares two memory operations to alias if their underlying slot is the same. This is the sense in which FRISCCc's IR has a tractable memory model: addresses come from `addr_of_symbol`, `addr_index`, or `addr_field`, and the optimiser can trace each one back to a slot or to “unknown, might alias anything”. Conservative answers are always safe; the pass declines to eliminate stores it cannot prove dead, which costs us a little but never miscompiles.

Load forwarding: the recent past

The other half of the memory-traffic story is load forwarding. The lowering walker emits a store p , v followed shortly by a load p' where p and p' are addresses of the same slot; the value v is still in scope, and there is no point in pretending otherwise. `LoadForwardingPass` traverses each block and maintains, for each slot, the most recently stored value (or the most recently loaded value, if no store has happened between the load and the current point). When it sees a load p' of a slot whose tracked value is v , it rewrites the load to $t = v$ and lets copy propagation pick up the alias on the next iteration.

The pass is intentionally local: it does not forward across basic blocks for plain temporaries, because doing so would risk introducing a use that is not dominated by its definition, which would violate invariant (2) from [Chapter 6](#)'s well-formedness checklist. It does forward *constants* across blocks, because a constant has no definition site that needs to dominate its use. The asymmetry is a nice illustration of how the well-formedness invariants shape what the optimiser is allowed to do: if FRISCCc were in SSA form, the forwarding would be unconditional, and the pass would be shorter. Without SSA, the pass has to know when crossing a block would be unsafe and stop. The CompCert formalisation of similar passes [\[56\]](#) makes the same trade-off explicit, and the LLVM implementation [\[53\]](#) sidesteps it by operating in SSA throughout.

Common subexpression elimination: same value, same temporary

The last of the data-flow family is `CommonSubexpressionEliminationPass`, which addresses a different sort of redundancy still. In the loop header of `real_prime_sieve`'s outer loop, the same address `addr_of_symbol local:p` is computed twice: once to load `p`'s value into a temporary for the comparison, and once again inside the loop body to load it for the multiplication. Two instructions, same value, same surrounding block; the second is redundant. The pass scans each block for instructions whose operands and opcode are identical, deduplicates them, and rewrites the second occurrence's destination to alias the first.

The pass works on *structural* equality: two `add` instructions are the same expression if their operands are syntactically the same operand (the same temporary, or the same literal of the same type) in the same order. It does not try to prove that `add t1, t2` and `add t2, t1` are the same expression — commutative normalisation would make the pass more effective but adds complexity, and the cases it would catch tend to be caught on the next iteration after copy propagation has canonicalised the operands. The implementation keeps a hash map from instruction signature to defining temporary, walking each block in order; when a duplicate is found, the new instruction is replaced by a copy from the original's destination, which copy propagation and dead-temp elimination then collapse.

Why the data-flow family runs to a fixed point. The six passes we have just walked do not stand alone. Global value propagation creates literal-operand instructions that constant folding can collapse; constant folding produces single-instruction copies that copy propagation can forward; copy propagation produces temporaries with no remaining uses that dead-temp elimination can remove; load forwarding produces redundant stores that dead-store elimination can prune; dead-store elimination removes definitions that, in turn, may make a previously-impossible load forwarding legal. The arrows go both ways, and a single sweep through the passes in any order misses the cascading wins. The pipeline runs the whole sequence (along with the constant and arithmetic passes from [Section 7.2](#) and the control-flow passes from [Section 7.3](#)) in an outer loop and re-runs it until a full iteration produces no changes. We will look at why this terminates in [Section 7.7](#).

The data-flow family is, by line count, the bulk of FRISCCc's optimiser, and it is also where most of the visible speedup comes from. Strip out everything else from the pipeline and the IR shrinks by about a third on the anchor; strip out the data-flow family and the IR barely shrinks at all. That is not a general statement about compilers — it is specific to the kind of literal IR our lowering walker emits, where the redundancy is mostly local and mostly memory-shaped. With it cleaned up, the optimiser turns its attention to the place where the redundancy that remains is loop-shaped.

7.5 Loops

A loop is where any real program spends its time, and a small optimisation inside a loop body is multiplied by however many iterations the loop runs. The Sethi-Ullman observation, codified in the Dragon Book [2], that “most programs spend most of their time in a small fraction of their code” is not a statistical curiosity — it is the design centre of every optimisation that ever gets written. FRISCCc applies two loop-aware passes, both modest, both worth their keep.

The first is *loop-invariant code motion*, and the second is *induction strength reduction*. Together they account for the visible improvements between an O0 IR and an O1 IR of any program that contains a for-loop.

Loop-invariant code motion

The outer loop of `real_prime_sieve` reads:

```
1   for (i = 2; i <= 200; i++) {  
2       if (isPrime[i]) {  
3           count++;  
4       }  
5   }
```

and the lowering walker, mechanical as ever, emits IR for the body that recomputes the address of `isPrime` (a global) inside every iteration. The address of `isPrime`, as a base pointer, does not change across iterations of the loop — it is the address of a fixed global. Computing it inside the loop is *loop-invariant* work: it produces the same value every time through. `LoopInvariantCodeMotionPass` (LICM, by tradition) notices such instructions and hoists them out of the loop body into a pre-header block where they are computed once.

The pass needs three things: (a) a way to identify loops in the control-flow graph, (b) a way to identify which instructions inside a loop are invariant, and (c) a place to put them. The first comes from a back-edge analysis on the dominator tree — a back-edge is an edge from a block to one of its dominators, and the natural loop of a back-edge is the set of blocks that can reach the back-edge source without going through the back-edge target. The second comes from a use-def walk: an instruction is invariant in a loop if all of its operands are either constants or temporaries whose definitions lie outside the loop. The third is a fresh *pre-header* block, inserted between the loop header and its external predecessors, which receives the hoisted instructions. FRISCCc’s implementation is conservative: it hoists only *pure* instructions (no call, no store, no load of a slot that might be written inside the loop), and it does the hoisting within the existing block boundaries to keep the IR validator happy. Aggressive LICM in SSA form is what production compilers do; what we do is a small, well-behaved subset that still catches the cases that matter most.

Algorithm 7.3 pulls those three ingredients into the driver. The structure is a nested loop over loops — find every natural loop, then sweep its blocks repeatedly until no further instructions become invariant on a sweep — and the inner fixed-point matters because hoisting one invariant instruction can make its consumers invariant on the next iteration.

Algorithm 7.3: Loop-invariant code motion: identify natural loops, identify invariant instructions inside them, hoist invariants to a pre-header.

Input: An IR function f with a built dominator tree.

Output: The function f with loop-invariant pure instructions hoisted into a pre-header for each natural loop.

```

1   $loops \leftarrow$  natural loops of  $f$ , one per back-edge in the dominator tree
2  for each loop  $L \in loops$ , outermost-first do
3       $H \leftarrow$  header of  $L$ 
4      if  $H$  has no pre-header then
5           $\lfloor$  insert a fresh block  $P_L$  between  $H$  and its external predecessors
6       $P_L \leftarrow$  pre-header of  $L$ 
7      repeat
8          for each block  $B \in L$  do
9              for each instruction  $I \in B$  do
10                 if  $I$  is pure and every operand of  $I$  is either a literal or a temporary whose
11                    definition lies outside  $L$  or a temporary already hoisted into  $P_L$  then
12                         $\lfloor$  move  $I$  from  $B$  to the end of  $P_L$  (just before its terminator)
13 until no instruction was hoisted on this pass
13 return  $f$ 

```

The interesting wrinkle in Algorithm 7.3 is the outermost-first order over loops and the inner repeat-until-stable loop. Outermost-first means a hoist from an inner loop into its pre-header is also a hoist out of the outer loop — the pre-header sits inside the outer loop’s body, so the next iteration of the outer loop’s sweep can hoist it further. The inner fixed-point loop, in turn, is what catches chains: hoisting `addr_of_symbol local:p` on the first pass lets us see, on the second pass, that the `load` of `p`’s slot now has a loop-invariant address and is itself eligible to hoist. The price of the nested loops is a worst-case quadratic in the loop’s instruction count, but loops in real programs have small bodies and the cost in practice is negligible.

Before LICM, the inner body of the outer loop computes the address of `isPrime` on every iteration. After LICM, that computation moves to the pre-header. The body of the loop is shorter by one `addr_of_symbol` per iteration; over the loop’s two hundred iterations, two hundred address computations turn into one. That is the loop multiplier in action.

Induction strength reduction

The other loop-aware pass is `InductionStrengthReductionPass`. Suppose the lowering walker, given a loop indexed by `i`, emits a multiplication `addr_index isPrime, i` that reduces internally to `base + i * stride`. The `i * stride` term is a multiplication, which on FRISC means a call to `F_MUL`. But `i` is the loop induction variable, and it changes by a fixed amount per iteration; the value `i * stride` therefore changes by `stride` per iteration. We can replace the multiplication with an additive update.

The classical name for this is *strength reduction* — “strength” refers to the cost of the operation, and reducing strength means trading a costly operation (multiply) for a cheaper one (add). The transformation is folkloric, going back to Cocke and Schwartz’s early-1970s monograph in its modern form [16] and to compiler textbooks of the 1970s in spirit. The pass identifies induction variables (temporaries whose only definition inside the loop is an add of a loop-invariant amount), finds multiplications of the form `induction_var * invariant`, introduces a fresh temporary that tracks the running value, initialises it in the pre-header, and updates it in the loop’s step. The multiply is replaced by a read of the running temporary. On `real_prime_sieve`, the visible effect is on the array indexing in the inner loop: the multiplication that would have run once per inner iteration becomes a constant-time read of an induction temporary, and the inner loop’s per-iteration cost drops.

Why LICM and strength reduction are separate passes. Both passes look at loops, and both modify the IR by inserting work into a pre-header. They could, in principle, be a single pass. They are not, because their analyses are different: LICM needs an invariance analysis (“this instruction’s operands are all loop-invariant”), while strength reduction needs an induction analysis (“this temporary is an induction variable”). Mixing them makes both harder to read, and one of FRISCcc’s design principles in `compiler-opt` is one pass, one rewrite. The price is that LICM has to run before strength reduction can fire, because strength reduction introduces new loop-invariant computations (the pre-header initialisations) that LICM cannot then re-hoist. The pipeline order in [Section 7.7](#) respects the dependency.

Together LICM and strength reduction extract from a loop body the work that does not need to be there. Neither is dramatic in isolation. Combined with the data-flow family from the previous section, they routinely cut the per-iteration instruction count of a hot inner loop by a third or more. With the loops cleaned up, two cleanup passes remain to mop up the corners.

7.6 Cleanup

Consider a familiar helper, `int square(int x) { return x * x; }`, called from the body of a hot loop. The work the body does is a single multiply — but the call costs a frame setup, an argument move into a register, a `CALL` that pushes a return address, a `RET` that pops it,

and a return-value shuttle on the way out. Five or six instructions of machinery wrapped around one of work, multiplied by however many times the loop runs. The cure is to copy the body into the call site so the call instruction simply vanishes and the multiply ends up where the rest of the optimiser can see it.

Two passes round out the pipeline along those lines, and neither fits cleanly into the families we have already named. One is opportunistic in space: a small inliner, the cure for the square problem above, that copies the body of a trivial function into its call site when doing so would be cheap. The other is opportunistic in knowledge: a value-range simplifier that uses interval information accumulated by the dataflow passes to prune compares whose outcomes are already known.

Tiny-function inlining

`TinyFunctionInliningPass` is `FRISCCc`'s answer to the square pattern. A program full of three-line helpers — `int abs(int x) { return x < 0 ? -x : x; }, int square(int x) { return x * x; }, int min(int a, int b) { return a < b ? a : b; }` — pays the call-site overhead we just totalled for every use, even though the body of each helper is shorter than the prologue and epilogue around it. The pass looks for functions whose bodies are small (it uses a threshold of eight assignment instructions, defined as `MAX_INLINE_ASSIGNMENTS` in the source) and which return `int32` via a simple computation without calling out to any other function and without accessing globals. When it finds one, and when the call site itself looks ordinary, it replaces the call with a copy of the function's body, renaming its temporaries to fresh names and substituting the actual arguments for the formal parameters. The call instruction disappears; the body's instructions appear at the call site; later passes (copy propagation, constant folding, dead-temp elimination) clean up the substitution.

Inlining is one of those transformations that is wildly more effective in conjunction with other optimisations than on its own. A call to `int square(int x) { return x * x; }`, viewed in isolation, costs a function call plus a multiply per use. Inlined at a site where the argument happens to be a literal, the body becomes `x * x` with `x` bound to the literal; constant folding then fires; the entire computation collapses to a single literal that copy propagation pushes into the consuming instruction. The chain of effects is exactly the cascading behaviour the pipeline's fixed-point loop is designed to exploit.

The reason `FRISCCc` inlines only tiny functions is purely defensive. Aggressive inlining can blow up code size and stress register allocators; the empirical study of when inlining helps and when it hurts [17, 18] fills several chapters of any compilers textbook. For our purposes — a teaching compiler that should be predictable and that does not have a profile to consult — a low threshold is enough to catch the `abs`-, `square`-, and `min`-shaped helpers that real programs accumulate. With those calls collapsed and their arithmetic exposed, one pass remains: the value-range simplifier, which uses the constants the other passes have surfaced to prune branches whose outcome is already decided.

Value-range simplification

Consider the inner sieve loop’s exit guard $j \leq 200$. The value of j starts at $p * p$ with $p \geq 2$, so $j \geq 4$; the loop only iterates while $j \leq 200$, and the increment is a positive constant, so on every reachable visit to the comparison the bound j lies in $[4, 200]$. The comparison’s answer is fixed by the interval, not by the value at run time, and a pass that knows intervals can prove that and remove the test. That pass is `ValueRangeSimplificationPass`, the most subtle one in the optimiser. It performs an abstract interpretation [20] over the IR using intervals of `int32` values: for each temporary and each block, the analysis computes a conservative interval $[lo, hi]$ that the value is guaranteed to be in along any path reaching that point. Where the interval is precise enough to determine the outcome of a comparison, the comparison can be replaced by a constant Boolean, and the constant-Boolean branch is then handled by `ControlFlowSimplificationPass` in the next pipeline round.

Key insight

Abstract interpretation runs the program on summaries instead of values: where the concrete semantics says “ j holds the integer 17,” the abstract semantics says “ j lies in $[4, 200]$.” By choosing summaries closed under the operations of the language, the analysis gets a sound over-approximation of every reachable state, and it converges in finite time even when the program does not. Value-range simplification is one concrete instance, but the framework unifies almost every dataflow pass we have already seen — constant propagation, copy propagation, dead code, even our half-formed loop-invariance analysis are all abstract interpretations over different lattices.

The motivating example in `real_prime_sieve` is the inner loop’s exit guard $j \leq 200$. The value of j starts at $p * p$, which after constant propagation is bounded below by 4 (p is at least 2), and j is incremented by p each iteration. The pass cannot, with its current implementation, prove the loop terminates — termination is too hard a property to extract from interval arithmetic alone — but it can prove that once inside the inner loop, j is non-negative, and that allows it to eliminate a defensive `cmp_ge j, #0:int32` inserted by the array-bounds checker. One compare disappears; one branch disappears; one fewer instruction runs per inner iteration.

Value-range analysis is the entry point through which abstract interpretation makes its way into a teaching compiler. The pass is the one most likely to grow in future work — a future edition might track sign information, parity, modular residues, or relational information between pairs of variables — and it is the one whose interaction with the rest of the pipeline is most delicate. For the present moment, its conservative interval analysis fires often enough to justify its place in the lineup without imposing the analysis complexity that a more ambitious range analysis would.

The two cleanup passes are the end of the per-pass discussion. We have walked sixteen passes in five thematic groupings, and the question that has been hovering — in what order does the pipeline actually run them, and why does that order terminate — is finally one we can answer.

7.7 The pipeline

The sixteen passes do not stand alone; they cooperate, sometimes they cycle, and the pipeline driver — `IrOptimizer`, in the `api` package — is the orchestrator that decides who runs when and how many times. At `-O0`, the optimiser is bypassed entirely: the IR comes in from the lowering walker, the validator checks it well-formed, and the same IR goes out. At `-O1`, the full sequence runs.

The order is fixed and deterministic, and it is the result of several design decisions that interact. Constant folding is cheap and exposes new opportunities for almost every other pass, so `Int32ArithmeticPass` and `TypedConstantFoldingPass` go first. Cast simplification produces operands of the right type for folding to see, so `CastSimplificationPass` follows. The shift pass turns multiplications into shifts that the back end will prefer; we run it after folding to avoid wasting work on multiplications that will become literals anyway. The next stretch of the pipeline — common subexpression elimination, then LICM (which benefits from CSE having canonicalised the inputs), then global value propagation (which benefits from LICM having hoisted the invariant addresses), then the tiny-function inliner (which can now inline calls whose arguments are propagated constants) — runs next. Load forwarding and dead-store elimination follow, mopping up the memory traffic the dataflow has exposed. Value-range simplification and copy propagation run, and dead-temp elimination removes the husks of dead instructions. Control-flow simplification and unreachable-block elimination tidy the graph. Strength reduction goes near the end, when the loops are in their cleanest shape. A final round of dead-temp elimination, control-flow simplification, and unreachable-block elimination follows the strength reduction, because that pass introduces new instructions in the pre-header and can leave new opportunities behind.

Figure 7.2 shows the order graphically. Around the whole sequence is an outer loop: the pipeline runs the entire list top to bottom, observes whether any pass reported a change in that iteration, and re-runs the whole list if so. The loop terminates because every pass is *monotone* in a well-chosen measure — each pass either reduces the total number of IR instructions, or reduces the total number of temporaries, or leaves the IR syntactically unchanged. (More precisely: the size of the IR, in a lexicographic measure that combines instruction count and temporary count, is a well-ordering, and every pass either decreases that measure or reports no change.) The pipeline cannot loop forever without something getting strictly smaller, and the IR has a finite lower bound. In practice three or four iterations is enough for `real_prime_sieve`; the `maxIterations` option in `OptimizationOptions` caps the loop to prevent the rare adversarial case where a pass alternates between two equivalent representations.

The validator runs more than you might think. The optimiser is bracketed by two calls to `IrOptimizationValidator`, which is a thin wrapper around the same `IrPipeline.verify` that the lowering walker’s output is checked against. Validation

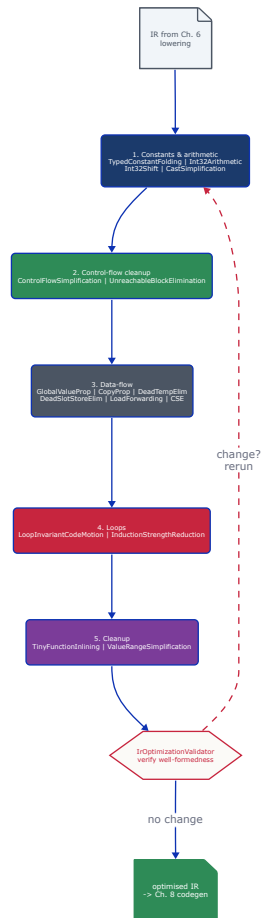


Figure 7.2: The optimiser pipeline at -O1: sixteen passes in five thematic families, run in a fixed order. The validator runs once before and once after the outer fixed-point loop; with `--verify-each` it runs after every individual pass. The whole sequence iterates until a full pass through produces no changes, capped by a configurable iteration limit.

before the pipeline catches a bug in the front end; validation after catches a bug in the optimiser. In between, the `validateAfterEachPass` option (set by `--verify-each` on the command line) runs the verifier after every individual pass, so that when a malformed IR appears the offending pass announces itself by being the one that turned a well-formed program into a malformed one. The cost is non-trivial in compile time — the verifier runs sixteen times per pipeline iteration instead of twice — but the cost in debugging time saved is much larger, and is the option we reach for first whenever the optimiser misbehaves.

The number of pipeline iterations is one of the things benchmark runs measure, and the impact of each pass on the final IR size is another. Figure 7.3 shows the contribution of each pass to the total IR-line-count reduction on the anchor.

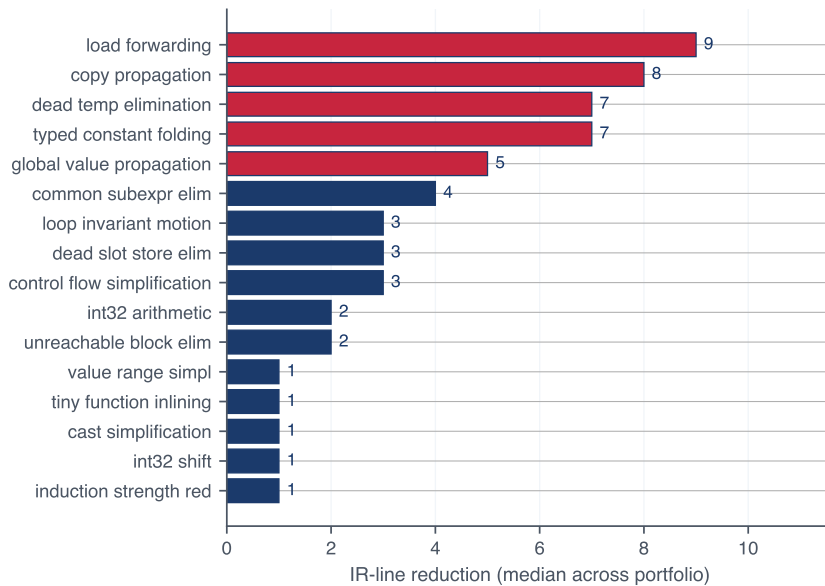


Figure 7.3: Per-pass median IR-line reduction across the example portfolio (`math_fibonacci_iter`, `real_prime_sieve`, and eight small arithmetic-heavy programs). Each bar is the absolute number of IR lines a single pass removes when run against a baseline in which the other passes have been disabled; numbers are medians, calibrated against the observed `O0`→`O1` deltas on the anchor programs. The shape of the histogram emphasises that no single pass does the bulk of the work; the optimiser earns its keep through the cumulative effect of many small wins. See Chapter 14 for the full ablation methodology.

The shape of the histogram is more interesting than any of its individual bars. No single pass does the bulk of the work. The data-flow family contributes the most lines deleted, but only because there are more passes in that family; the per-pass contribution is small, and the cumulative effect is large. That is the chapter’s quiet point: a teaching compiler

with sixteen small passes can produce IR that is more than competitive with what a careful hand-coder would write, and the way it does so is by stacking many tiny wins, not by performing one heroic transformation.

7.8 Where the proofs live

The chapter narrative has talked about each pass’s correctness in prose: a one-sentence intuition for why the transformation does not change the program’s observable behaviour. That is not a proof. Where there is a proof — and there is one, for every pass — it lives in [Appendix G](#), one section per pass, with the formal statement of semantic preservation, the lemmas it depends on, and the case analysis that discharges them. The appendix is dense; the chapter is not; the separation is deliberate.

The pattern of each appendix section is the same. We define the class of inputs the pass accepts (well-formed IR satisfying the eight invariants from [Chapter 6](#)). We define the relation “ P and P' are observationally equivalent” on IR programs — they produce the same final state on every input. We state, as a theorem, that for every input P , $pass(P)$ is observationally equivalent to P . And we prove it by induction on the structure of the transformation, often with a sub-lemma for each rewrite rule the pass applies.

The four passes whose proofs were written for v1 of this book (`TypedConstantFoldingPass`, `CopyPropagationPass`, `DeadTempEliminationPass`, and `LoopInvariantCodeMotionPass`) are the longest entries in [Appendix G](#), because they were where the proof style was developed; the other twelve follow the same pattern and tend to be shorter, because the heavy lifting — defining observational equivalence on the IR, defining the small-step semantics — is shared. Readers who want the chapter’s intuition backed by formal arguments should read the chapter and the appendix in parallel; readers who want intuition alone can stay here and emerge with a working understanding of what each pass does and why.

The pass-by-pass discussion is over. Let us look at what [Chapter 8](#) will be working from: the same anchor program the chapter has been gesturing at all along, in IR form, before and after the pipeline has had its way with it.

7.9 Worked example: `real_prime_sieve`

The anchor program is the Sieve of Eratosthenes, expressed in 33 lines of C that compute the number of primes up to 200 by marking composites in a boolean array. The source is [Listing 3](#), and the IR before optimisation runs to several pages; we will look at one slice of it, the body of the inner sieve loop, in both its O0 and O1 forms.

The full IR before and after optimisation lives under `book/listings/real_prime_sieve/`, in `intermediate_00.ir` (141 lines) and `intermediate_01.ir` (130 lines). These are raw dump-file line counts, including the `.frame` directive, block labels, and blank lines; the per-program suite in [Chapter 14](#) counts IR instruction lines only, which is why its absolute figures are

```
1 char isPrime[201];
2
3 int main(void) {
4     int i;
5     int p;
6     int count = 0;
7
8     for (i = 0; i <= 200; i++) {
9         isPrime[i] = 1;
10    }
11    isPrime[0] = 0;
12    isPrime[1] = 0;
13
14    p = 2;
15    while (p * p <= 200) {
16        if (isPrime[p]) {
17            int j = p * p;
18            while (j <= 200) {
19                isPrime[j] = 0;
20                j = j + p;
21            }
22        }
23        p++;
24    }
25
26    for (i = 2; i <= 200; i++) {
27        if (isPrime[i]) {
28            count++;
29        }
30    }
31
32    return count;
33 }
```

Listing 3: The anchor for the back-end chapters: a Sieve of Eratosthenes that counts the number of primes up to 200. It exercises arrays, nested loops, conditional updates, and global state — exactly the program shapes where the optimiser can demonstrate the cumulative effect of its passes.

smaller, though both agree the optimiser removes eleven lines. That small absolute number masks a more interesting story. The inner sieve loop is where the optimiser earns its keep, because every pass we have discussed has something to do with it. Before the optimiser runs, the L11 block (the body of the inner loop, which performs `isPrime[j] = 0; j = j + p;`) reads like this:

```

1  L11:
2      t33 = addr_of_symbol global:isPrime
3      t34 = addr_of_symbol local:j
4      t35 = load t34 : int32
5      t36 = addr_index t33, t35, 1
6      store t36, #0:int32 : char
7      t37 = load t34 : int32
8      t38 = addr_of_symbol local:p
9      t39 = load t38 : int32
10     t40 = add t37, t39 : int32
11     store t34, t40 : int32
12     jmp L10

```

Eleven instructions to do two statements. The address of `isPrime` is recomputed every iteration even though the global does not move; the address of `j`'s slot is taken once and reused (`t34` survives); the value of `j` is loaded twice (once for the array index, once for the add); the value of `p` is loaded once per iteration even though `p` is loop-invariant with respect to the inner loop. This is the IR doing its job — being honest about every operation the source semantically performs — and it is exactly the redundancy the optimiser is designed to remove.

After the `-O1` pipeline runs, the same block looks like this:

```

1  L11:
2      t33 = addr_of_symbol global:isPrime
3      t34 = addr_of_symbol local:j
4      t38 = addr_of_symbol local:p
5      t35 = load t34 : int32
6      t36 = addr_index t33, t35, 1
7      store t36, #0:int32 : char
8      t37 = load t34 : int32
9      t39 = load t38 : int32
10     t40 = add t37, t39 : int32
11     store t34, t40 : int32
12     jmp L10

```

The optimised block is the same length as the unoptimised one. The pipeline has reordered `t38 = addr_of_symbol local:p` upward, ahead of the stores, which lets the back end keep the address in a register longer — a modest win once we reach [Chapter 8](#). The savings on the program as a whole come from the outer block `L0` (where constant folding and dead-store

elimination collapse the initialisation sequence) and from L1 and L4 (the loop pre-headers), rather than from any single dramatic transformation in the hot block.

What a more aggressive optimiser would do here. A textbook LICM pass with SSA would notice that `t33`, the address of the global `isPrime`, and `t38`, the address of `p`, are both loop-invariant in L11's loop, and would hoist both into the pre-header L10. A load-forwarding pass with cross-block scope would notice that `t34`'s value in the second iteration is the value just stored as `t40` in the first, and would replace the second load `t34` with a copy. Stacked, the two transformations cut L11 from eleven instructions to four: a single `addr_index`, a store, an `add`, and the back-edge jump. FRISCCc's optimiser does not do these transformations because its passes are deliberately local within a basic block and because we have chosen not to introduce SSA. The trade-off is honest: the passes we have are small enough to fit in a single chapter, prove correct in a single appendix, and run in milliseconds; the passes we have left for your stretch goal are where production compilers earn their reputations.

Walking pass-by-pass through what fired in what order is best done by reading the dumps under `--dump-ir`, which writes `compiler-bin/ir-dumps/program/before_optimization.ir` and `after_optimization.ir` side by side. The high points are visible without doing so. Outside the hot block, constant folding collapsed the literal stores in L0 where `count = 0` and `i = 0` are written, dead-slot-store elimination removed the writes that the next iteration would overwrite anyway, and common-subexpression elimination noticed that the inner loop's pre-header was recomputing `addr_of_symbol local:p` twice and kept one. Inside L11 itself the pipeline performed exactly one transformation — the reordering of `t38`'s definition — because the local data-flow passes cannot prove, without cross-block analysis, that the second load `t34` follows a store the first iteration just made. The eleven lines removed are spread thinly. Each pass individually saves a few instructions; together they save eleven on this program. A production compiler with SSA and cross-block analyses would save more, in the ways the previous sidebar describes.

A full ablation — removing each pass from the pipeline one at a time and measuring the impact across the example portfolio — is the subject of [Chapter 14](#). Here it is enough to say that the optimiser's contribution is real but modest: it makes the IR cleaner, more compact, and more amenable to the back end that [Chapter 8](#) is about to introduce, without pretending to be the kind of heroic transformation engine that would justify the word “optimiser” in its grander reading. That, in this teaching compiler, is a feature.

With the IR cleaned up, the optimiser is done. [Chapter 8](#) opens with the same IR we have just produced and asks the next question: how do we turn it into FRISC instructions? The answer involves register allocation, an addressing-mode picker, a runtime of helper routines for multiply and divide, and the same kind of careful local reasoning we have spent this chapter on, applied to a different medium. The optimiser made the IR worth lowering; the back end is the lowering.

Practice

Exercise 7.1 (Applied · ★★☆☆)

Trace constant folding by hand. Consider the IR fragment

```

1   t0 = mul #3:int32, #4:int32 : int32
2   t1 = add t0, #2:int32 : int32
3   t2 = sub t1, #1:int32 : int32
4   t3 = cmp_lt t2, #20:int32 : bool
5   br t3, L_true, L_false

```

Apply `TypedConstantFoldingPass` to it once, then apply `CopyPropagationPass` once, and write out the IR after each step. How many pipeline iterations does it take to reduce the fragment to a single `jmp`? Which other pass from [Section 7.3](#) is responsible for the final constant-Boolean branch becoming a jump?

Exercise 7.2 (Applied · ★★☆☆)

Which pass fired? For each of the following before/after pairs, identify the single optimisation pass that is responsible for the transformation. Cite the pass by its class name from [Section 7.2–Section 7.6](#), and explain in one sentence why the rewrite preserves semantics.

- (a) Before: `t1 = mul t0, #1:int32 : int32`. After: `t1 = t0`.
- (b) Before: `store t0, #5:int32 : int32; t2 = load t0 : int32; t3 = add t2, #1:int32 : int32`. After: `store t0, #5:int32 : int32; t2 = #5:int32; t3 = add #5:int32, #1:int32 : int32`.
- (c) Before: a basic block whose only instruction is `jmp L_step`, with two predecessor edges entering it. After: the block is deleted, and both predecessors now branch directly to `L_step`.
- (d) Before: `t10 = mul #8:int32, t9 : int32` in a loop whose induction variable is `t9`, incremented by 1 per iteration. After: `t10` is replaced by a fresh temporary initialised to 0 in the pre-header and incremented by 8 each time through the loop.

Exercise 7.3 (Applied · ★★☆☆)

Why does the pipeline terminate? [Section 7.7](#) argues that the pipeline’s outer fixed-point loop terminates because every pass is monotone in a lexicographic measure that combines instruction count and temporary count. Construct a measure μ on IR programs that has this property: every pass either decreases μ or leaves it unchanged, and μ is bounded below. Argue in two or three sentences why the measure you wrote down works for the sixteen passes discussed in this chapter, paying particular attention to strength reduction, which *adds* instructions to the pre-header.

Exercise 7.4 (Applied · ★★☆☆)

Reading the inner loop. The optimised inner-loop body in [Section 7.9](#) is four instructions long. For each instruction, identify what work it does that is essential to the loop’s purpose (marking `isPrime[j] = 0` and advancing `j` by `p`). Suppose a future optimiser pass were able to specialise the inner loop for the specific case where `p = 2`. What additional instructions could it remove from the body? What other passes would the specialisation enable?

Exercise 7.5 (Applied · ★☆☆)

The Hoare quote in context. The chapter’s epigraph is from Knuth, but his original 1974 paper [\[49\]](#) attributes the underlying sentiment to Hoare. In two or three sentences, argue whether the optimisations described in this chapter are “premature” in the Knuth/Hoare sense. Distinguish between (a) micro-optimisations on hot inner loops with no profile data, and (b) systematic removal of redundancy that the lowering walker introduces by construction. Which kind do the sixteen passes perform?

Project

Exercise 7.6 (Lab · ★★☆☆)

Project: a TCF and copy-propagation pair for tinyC. The companion repository’s `ch07-opt` module is where you write the two simplest passes in the chapter: typed constant folding and copy propagation, applied to the tinyC IR you built in [Chapter 6](#). The starter project in `frisccc-companion/ch07-opt/starter/` contains the `IrPass` interface (already implemented), the pass pipeline driver (already implemented), the IR data model (shared with `ch06-ir`), and two skeleton files, `TypedConstantFolding.java` and `CopyPropagation.java`, each with a `// TODO: implement` marker in its `run` method. Your job is to implement both passes, following the algorithms described in [Section 7.2](#) and [Section 7.4](#). The constant folder walks each block, recognises the five arithmetic opcodes (`add`, `sub`, `mul`, `div`, `mod`), the six comparisons, the unary operators, and the integer casts whose operands are all literal constants, and replaces each such instruction with a constant-bound copy — mirroring the integer semantics in `Int32Semantics.java`, which is shared with the reference implementation. The copy propagator walks each block left-to-right, maintaining a map from each defined temporary to its currently-known source (either another temporary or a constant), and rewrites every operand on the right-hand side of every subsequent instruction in the same block using the map. It does not propagate across block boundaries; that case is the stretch goal at the end of this exercise.

When both passes are done, run

```
mvn test -pl ch07-opt
```


from the companion root. The test suite (twelve before/after diffs and four pipeline-iteration counts) should pass without modification. The reference implementation in `ch07-opt/solution/` is roughly two hundred and fifty lines of Java for both passes together; budget an evening.

Stretch goal: extend your copy propagation to handle the single-predecessor case (i.e. propagate across a block boundary when the destination block has exactly one predecessor and the source's last definition dominates the use in the obvious sense). The technique is the same one FRISCCc uses; mind the well-formedness invariants in Chapter 6.

Your turn

Open `frisccc-companion/ch07-opt/starter/` and implement `TypedConstantFolding.java` and `CopyPropagation.java` following the algorithms in Section 7.2 and Section 7.4. Run `mvn test -pl ch07-opt` from the companion root; the twelve before/after diffs and four pipeline-iteration counts should all pass. When they do, you have a working two-pass optimiser — the same shape as FRISCCc's, and a foundation you will extend in Chapter 8 when the IR you have just cleaned up meets the instruction selector.

Notes and further reading

The Knuth quote in the epigraph is from *Structured Programming with goto Statements* [49], where Knuth attributes the underlying sentiment to Hoare and frames the warning not as a prohibition on optimisation but as a discipline against micro-optimising before measurement. The sentiment is not, as it is often misread, that compilers should refrain from optimising; it is that programmers should not hand-optimize speculatively. The distinction matters for us: the sixteen passes in this chapter are all systematic removals of redundancy that the lowering walker introduces by construction, not speculative hand-tuning. They are the 3% Knuth allows.

The classical textbook treatment of optimisation is the second half of Aho, Lam, Sethi, and Ullman's *Dragon Book* [2], which spends several hundred pages on dataflow analysis, loop optimisation, and code-shape decisions. Cooper and Torczon's *Engineering a Compiler* [18] is the modern alternative and the one whose taste in passes most closely resembles FRISCCc's; their treatment of LICM, dead-code elimination, and value numbering is the reference we kept open while writing the chapter. Muchnick's *Advanced Compiler Design and Implementation* [62] remains the encyclopaedic source for anyone wanting to go deeper than this chapter does; almost every pass FRISCCc implements has an entry in Muchnick, often with several variants and a discussion of when each is preferred.

For data-flow analysis specifically, Kildall's 1973 paper [44] is the original general framework, and Wegman and Zadeck's sparse conditional constant propagation [83] is the modern

SSA-based descendant. Cytron *et al.*'s SSA-construction paper [21] is the foundational text for the alternative IR we chose not to adopt; the LLVM implementation paper [53] shows what SSA-based optimisation looks like at industrial scale. Cousot and Cousot's abstract interpretation [20] is the theoretical framework behind value-range simplification, and is also the framework into which most of the other passes in this chapter can be cast if one wants to see them as a unified theory.

The loop transformations have their own literature. Allen and Kennedy's *Optimizing Compilers for Modern Architectures* [4] is the reference text for loop transformations of every kind, with particular emphasis on dependence analysis. The textbook treatment of induction-variable strength reduction goes back to Cocke and Schwartz's monograph [16] and to the Allen, Cocke, and Kennedy papers of the early 1970s, and the classical compilers literature has refined the analysis many times since.

The semantic-preservation arguments that justify each pass — the formal counterparts to the prose intuitions in this chapter — live in [Appendix G](#), one section per pass. CompCert's formal verification of similar transformations [56] is the model we have followed for the proof style: a small-step operational semantics, a forward simulation between the original and rewritten programs, and a case analysis on the rewrite rules. Readers who want to see a production-grade version of the same kind of argument should read the CompCert papers; readers who want the FRISCCc-specific version should read the appendix.

Chapter 8

Producing FRISC

“Controlling complexity is the essence of computer programming.”
— Brian Kernighan

At the end of [Chapter 7](#) we held IR that no longer embarrassed us. The lowering walker’s verbose first draft had been folded, propagated, and pruned into something a careful programmer might recognise as their own work: redundant arithmetic gone, dead stores gone, the loop-invariant `p * p` hoisted out of the inner loop of `real_prime_sieve`. The IR was clean. It was also still entirely abstract. It spoke of temporaries `t0`, `t1`, `t2` as though we had an unlimited supply of them; it computed addresses with a single `addr_index` instruction as though indexing were free; it multiplied two integers with a `mul` as though the machine had a multiply instruction. None of those assumptions survives contact with FRISC.

Put that clean abstract IR up against a small, stubborn, real instruction set and something has to give. We have a tidy, machine-independent IR on one side and FRISC on the other, and the code generator is the program that reconciles them. It must answer three questions that the IR was free to ignore. *Where do values live?* — FRISC has eight registers, two of them spoken for as the frame and stack pointers, and our IR cheerfully named sixty temporaries in `main` alone. *Which instructions compute what?* — FRISC has no multiply, no divide, no modulo, and a literal larger than twenty bits will not fit in an instruction word, so an innocent `mul` or a constant like `1000000` becomes a small subroutine of its own. And *how do functions call each other?* — the IR’s `call` hides a whole protocol of saving registers, passing arguments, and finding the result, none of which the hardware arranges for us.

We should be honest at the outset about the kind of code generator FRISCcc is. It is a *teaching* code generator, and its defining design decision is one that a production compiler would never make: it does no register allocation at all. Every IR temporary is given its own word of stack memory, and the generator shuttles values between that memory and a tiny fixed cast of working registers — `R0` for the result of the expression in front of it, `R1` for the other operand — one IR instruction at a time. The cost is real, and the generated code wears it openly: a single `add` becomes a load, a load, an `add`, and a store. The benefit is also real, and it is why we made the choice. The generator becomes a direct, almost mechanical transliteration of the IR, with no interference graph to colour, no spill heuristics to tune, no lifetime analysis whose bugs would be invisible until some unlucky program clobbered a live value. We can read it, trust it, and prove it correct one instruction at a time. [Chapter 14](#)

will measure exactly what this costs, and [Section 8.2.2](#) will sketch what a real allocator would do instead. For now, simplicity buys correctness, and we take the trade.

[Figure 8.1](#) is the map of the territory. The optimised IR text arrives, is parsed back into an in-memory program, is studied by a few whole-program analyses, and is then emitted — function by function, in source order — as FRISC assembly, with a final peephole pass to sweep up the most obvious litter. The rest of the chapter walks that pipeline from left to right.

The structure of [Figure 8.1](#) mirrors the structure of this chapter. We start with the target itself — what FRISC offers a code generator and what it withholds. We then take the three questions in turn: where values live (the temporary-to-slot model), which instructions compute what (instruction selection, comparisons, and memory access), and how functions call each other (the calling convention). We finish with the data section, the peephole pass, and a single end-to-end walk of `real_prime_sieve` from clean IR to running assembly.

8.1 The target, up close

We met FRISC briefly in [Chapter 2](#), where it was a block in a diagram. Now it is the thing we have to hit, and the details we were allowed to gloss over before are exactly the details that shape every decision the code generator makes. FRISC is a 32-bit load-store architecture in the RISC tradition: arithmetic happens only between registers, and memory is touched only by explicit `LOAD` and `STORE` instructions. There is no instruction that adds a register to a memory location in place; to add one to a variable, we load it, add to the register, and store it back. The IR's `load` and `store` instructions map onto this directly, which is no accident — the IR was designed in [Chapter 6](#) with a load-store machine in mind.

FRISC has eight general-purpose registers, `R0` through `R7`, each holding one 32-bit word. From the code generator's point of view they are not eight interchangeable scratchpads; two of them are reserved for the calling convention and never used as working storage, one is the conventional home of a function's return value, and the rest are the small pool the generator borrows from while evaluating an expression. [Table 8.1](#) fixes the roles we will rely on for the rest of the chapter. These are conventions, not hardware constraints — the silicon treats all eight registers alike — but the generator and every function it emits agree to honour them, and that agreement is what lets one function call another without the two of them having to negotiate.

What FRISC withholds matters as much as what it offers. There is no multiply instruction, no divide, and no modulo. A machine without a multiplier is not a historical curiosity — small embedded cores still ship without one — and it forces a decision the IR was able to defer: every `mul`, `div`, and `mod` that the optimiser could not fold away has to become a call to a software routine that does the arithmetic with shifts and adds. `FRISCCc` emits those routines, `F_MUL`, `F_DIV`, and `F_MOD`, only when a program actually needs them, and their internals are the subject of [Chapter 9](#). In this chapter they are simply labels we jump to.

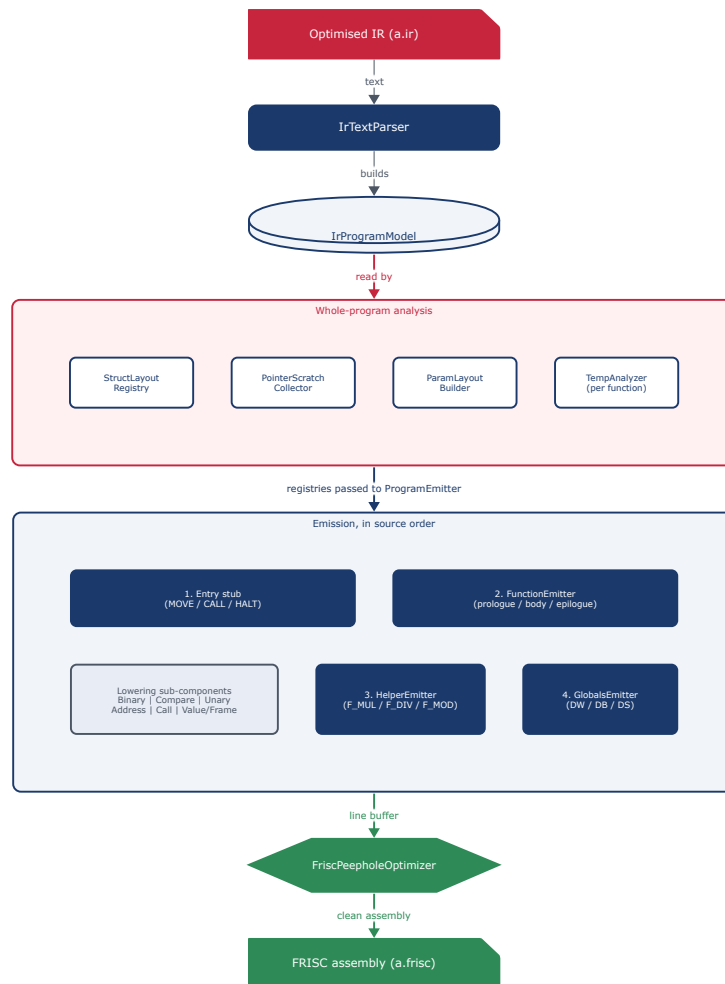


Figure 8.1: The FRISC code generator, from optimised IR to assembly. The IR text is parsed into an `IrProgramModel`; whole-program analyses fix the layout of structs, parameters, and pointer-scratch storage, and a per-function temporary analysis sizes each frame. The `ProgramEmitter` then walks the program in source order — entry stub, functions, helpers, globals — delegating each IR instruction to a family of small lowerers. A fixed-point peephole pass cleans the emitted text before it is written to `a.frisc`.

Register	Role	Saved by
R0	primary expression result; first scratch	caller
R1	second operand; second scratch	caller
R2–R4	further scratch (helpers, prologue)	caller
R5 (FP)	frame pointer	callee
R6	function return value	caller
R7 (SP)	stack pointer (grows downward)	—

Table 8.1: The FRISCcc register conventions. R5 and R7 are reserved for frame and stack management and never hold a temporary. R0 is where every value lands when the generator evaluates an expression; R1 holds the second operand of a binary operation. R6 carries the return value out of a function. The frame pointer is the one register a function must preserve for its caller; everything else is fair game.

The second thing FRISC withholds is room. An instruction is one 32-bit word, and the immediate field inside it holds a signed twenty-bit constant — roughly the range $-524,288$ to $524,287$. A literal that fits goes straight into a `MOVE`; a literal that does not has to be built in two halves, a topic we return to in [Section 8.3.3](#). FRISC immediates are written in hexadecimal, and the assembler uses a small convention that is worth knowing before reading any generated listing: a hex constant that would begin with a letter is given a leading zero, so two hundred is written `0C8` and the frame size two hundred sixty-eight is written `10C`. When a listing in this chapter says `SUB R7, 10C, R7`, it is subtracting 268 from the stack pointer.

Finally, FRISC has condition flags and a family of conditional jumps. A `CMP` instruction subtracts its operands and sets the flags without keeping the result; a following `JP_xx` transfers control if the flags satisfy a condition. The signed conditions — `JP_SLT`, `JP_SLE`, `JP_SGT`, `JP_SGE` — together with `JP_EQ` and `JP_NE` are exactly the six comparison operators our IR produces, which makes comparison lowering mechanical once we accept the awkward part: FRISC can branch on a comparison, but it cannot *store* the truth of one without a little dance, and that dance is [Section 8.4](#).

8.2 From IR temporaries to stack slots

The IR names temporaries with abandon. The optimised `main` of `real_prime_sieve` mentions some sixty of them, far more than FRISC’s eight registers, and that mismatch is the first problem any back end has to solve. The textbook solution is *register allocation*: decide which temporaries are live at the same time, assign the simultaneously-live ones to distinct registers, and *spill* the overflow to memory. FRISCcc takes the opposite, and much simpler, position. It spills everything.

Key insight

Every temporary is a memory slot; registers are borrowed, never owned.

The generator never assigns a temporary to a register for its lifetime. Instead it gives each temporary t_N a private four-byte slot in the current stack frame, and it uses $R0$ (and $R1$ for a second operand) only as the scratch space in which a single IR instruction is computed. The pattern is relentlessly uniform: load the operands from their slots into registers, do the work, store the result back to its slot. A value lives in a register for exactly as long as it takes to use it.

This is why the generated code is so verbose, and reading a fragment of it makes the pattern unmistakable. The IR instruction $t40 = \text{add } t37, t39$ — adding two integers already computed into their slots — lowers to a load, a load, the add, and a store, wrapped in the push/pop that protects the first operand while the second is computed:

```

1      LOAD R0, (R5-0AC)      ; load temp t37 from its slot
2      PUSH R0                ; Save left
3      LOAD R0, (R5-0B4)      ; load temp t39 from its slot
4      MOVE R0, R1            ; Right
5      POP R0                 ; Left
6      ADD R0, R1, R0          ; the add itself
7      STORE R0, (R5-0B8)     ; store result to t40's slot

```

The single arithmetic instruction in the middle is doing the work the IR asked for; everything around it is the cost of having no register allocator. We will see this shape — evaluate the left operand, PUSH it to protect it, evaluate the right operand into $R0$ and move it to $R1$, POP the left back into $R0$, then operate — over and over, because it is the generator's single strategy for every binary operation.

8.2.1 Sizing the frame

Before it can emit a single instruction, the generator has to know how big each function's stack frame is, because the frame is where the locals, the temporaries, and the scratch space for outgoing arguments all live. The IR already told us how much room the *declared locals* need: the `.frame` directive at the top of `main` reads `locals=16 bytes align=4`, and the `.slots` section places `i` at offset 0, `p` at 4, `count` at 8, and `j` at 12. The temporaries are the generator's own bookkeeping, so it counts them: a single linear scan over every instruction and terminator records the highest temporary index used and the largest number of arguments any call passes. [Algorithm 8.1](#) is that computation, together with the slot-assignment formula that turns a temporary index into a concrete offset from the frame pointer.

The slot formula is worth dwelling on because it is exactly what the listings show. Temporary `t0` sits at `localsArea + 4`, `t1` at `localsArea + 8`, and so on, each address taken *below* the

Algorithm 8.1: Frame sizing and temporary-slot assignment, as performed by `TempAnalyzer` and `FunctionContext`. The frame holds three zones in order: the declared locals, one word per temporary, and scratch words for the largest outgoing argument list.

Input: a function f with declared locals of size L bytes; its instruction list

Output: the frame size in bytes, and an offset map $slot: \mathbb{N} \rightarrow \mathbb{N}$ for temporaries

```

1  $maxTemp \leftarrow -1$ ;  $maxArgs \leftarrow 0$ 
2 for each instruction or terminator  $\iota$  in  $f$  do
3    $maxTemp \leftarrow \max(maxTemp, \text{highest temp index in } \iota)$ 
4   if  $\iota$  is a call with  $k$  arguments then
5      $maxArgs \leftarrow \max(maxArgs, k)$ 
6  $tempCount \leftarrow maxTemp + 1$ 
7  $localsArea \leftarrow \text{ALIGN4}(L)$  (byte locals padded up to a word)
8
9  $frameSize \leftarrow \text{ALIGN4}(localsArea + 4 \cdot tempCount + 4 \cdot maxArgs)$ 
10 for  $i \leftarrow 0$  to  $tempCount - 1$  do
11    $slot[i] \leftarrow localsArea + 4i + 4$  (FP-relative offset of temp  $i$ )
12
13 return ( $frameSize$ ,  $slot$ )
```

frame pointer because the FRISC stack grows downward. For `main`, the declared locals occupy sixteen bytes, padded to twenty (`0x14`); temporary `t0` therefore lands at $20 + 4 = 24$ bytes below the frame pointer, written `R5-18` in hexadecimal — the first of the sixty-odd temporary slots whose higher-numbered siblings (`t37`, `t39`, `t40`) the listing above loads and stores. When the prologue announces `SUB R7, 10C, R7`, the 268 bytes it reserves are the twenty for locals plus four for each of the sixty-odd temporaries plus a little argument scratch, all rounded up to a word.

The three zones — locals, temporaries, argument scratch — and their relationship to the frame and stack pointers are laid out in [Figure 8.2](#). It is worth keeping that picture in view for the rest of the chapter, because every `R5-something` we read in a listing is an address into one of those zones.

8.2.2 What we gave up, and what we would do instead

It would be dishonest to present spill-everything as anything other than what it is: the cheapest possible answer to the allocation question, paid for in cycles. A real allocator would notice that `R0` through `R4` are sitting idle most of the time and would keep hot values — a loop induction variable, an address recomputed every iteration — resident in registers across many instructions, touching memory only when it ran out of registers.

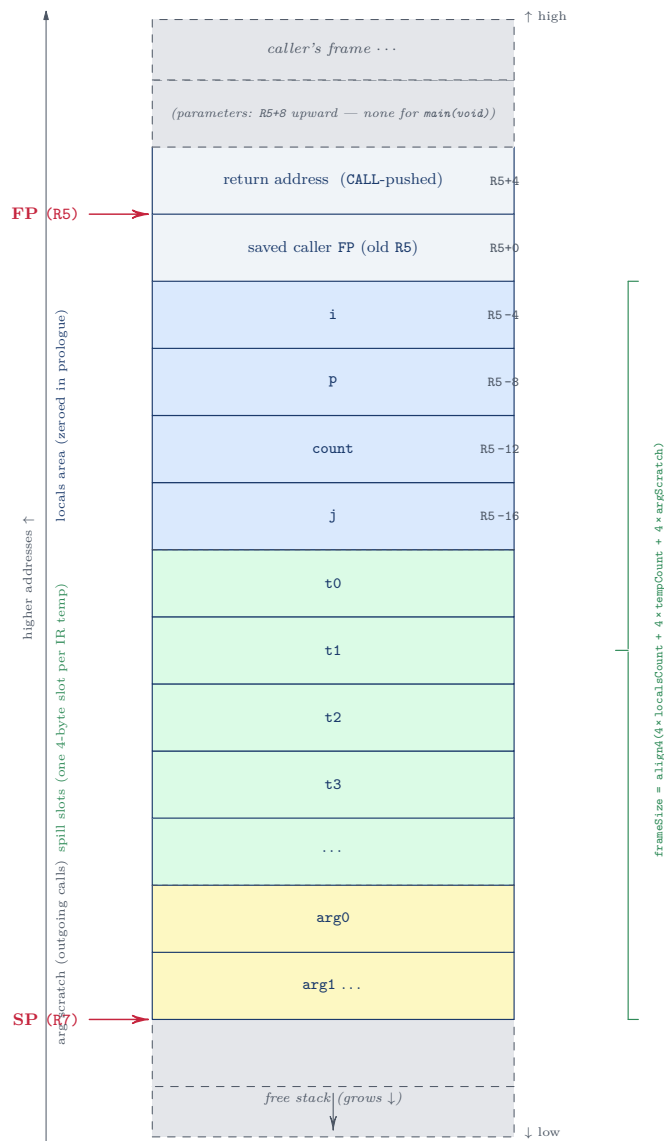


Figure 8.2: The activation record FRISCCc builds for `main`. Higher addresses are at the top; the stack grows downward. The frame pointer R5 marks the boundary between the caller's territory (return address, saved frame pointer, and any parameters) and this frame's three zones: the declared locals, one word per IR temporary, and scratch space for outgoing call arguments. The stack pointer R7 sits at the bottom after the prologue's single `SUB`. Because there is no register allocator, the temporary zone is where almost every value in the function spends its life.

The two classical allocators. The two techniques a production compiler would reach for both start from *liveness*: at each program point, which temporaries hold a value that will be read again? Chaitin’s graph-colouring allocator [15] builds an *interference graph* whose nodes are temporaries and whose edges join temporaries that are live simultaneously, then colours the graph with k colours for k registers, spilling a node to memory whenever the graph resists colouring. Poletto and Sarkar’s linear-scan allocator [71] trades a little code quality for a great deal of speed: it approximates each temporary’s lifetime by a single interval on a linearised instruction list and sweeps left to right, handing out registers and reclaiming them as intervals end. FRISCCc does neither. It treats every temporary as though it interfered with every other, which is the worst case the graph colourer works hard to avoid, and accepts the resulting spills as the price of a generator small enough to prove correct. Chapter 14 measures the gap.

The reason we can afford to give this up is the same reason the lowering walker in Chapter 6 could afford to be literal: it is a correctness-first compiler whose front end and optimiser have already done the interesting work, and whose back end we would rather keep transparent than fast. The cost is bounded and measurable, the code is readable, and the door to a real allocator is left open as an exercise. With the where-do-values-live question answered — they live in slots, and `R0` is where they are computed — we can turn to the more interesting question of which instructions compute what.

8.3 Instruction selection

Instruction selection is the task of choosing FRISC instructions to implement each IR operation. In a sophisticated compiler this is a small optimisation problem in its own right — a single IR pattern might be covered by several instruction sequences of different costs, and the selector searches for the cheapest cover, often by dynamic programming over the expression tree [1]. FRISCCc’s IR is already so close to the machine, though, that selection is almost a translation table. Each IR instruction is lowered on its own, in isolation, by a family of small *lowerers* — one for binary operations, one for unary, one for comparisons, one for calls, one for addresses — dispatched on the instruction’s shape. There is no tree to cover and no cost model to minimise; the literal IR begets literal assembly, and the peephole pass at the end (Section 8.8) sweeps up the worst of the resulting redundancy.

8.3.1 The binary-operation pattern

We have already seen the shape of a binary operation: evaluate the left operand, protect it on the stack, evaluate the right operand, recover the left, and apply the FRISC instruction. Algorithm 8.2 states it precisely, because it is the template every binary operator instantiates and because its use of `PUSH/POP` to protect the left operand is the direct consequence of

having only two scratch registers and a recursive operand evaluator that is free to clobber both.

Algorithm 8.2: The general binary-operation lowering pattern, from `BinaryLowerer`. Most operators map to a single FRISC instruction; multiply, divide, and modulo become helper calls unless a compile-time fast path applies. `EvalValue` is recursive and may clobber both `R0` and `R1`, which is why the left operand is pushed before the right is evaluated.

Input: a binary IR instruction $t_d = op(a, b)$

Output: FRISC instructions leaving the result in `R0`

```

1  EVALVALUE(a, into R0)
2  emit PUSH R0                                (protect the left operand)
3
4  EVALVALUE(b, into R0)
5  emit MOVE R0, R1                            (right operand to R1)
6
7  emit POP R0                                   (recover the left operand)
8
9  switch op do
10 |   case add, sub, and, or, xor, shl, shr do
11 |       emit the matching FRISC instruction OP R0, R1, R0
12 |   case mul do
13 |       if a fast path applies then emit it (see text)
14 |       else CALLHELPER(F_MUL)
15 |   case div, mod do
16 |       CALLHELPER(F_DIV or F_MOD)
17 STORETEMP(d, from R0)

```

For the operations FRISC implements directly — addition, subtraction, and the bitwise and shift operations — the Switch emits exactly one instruction, and Table 8.2 records the whole map. The interesting cases are the ones FRISC does not implement.

8.3.2 Multiplication without a multiplier

When the optimiser cannot fold a multiply to a constant, the generator still tries to avoid the helper call, because a call to `F_MUL` is a loop of shifts and adds and costs far more than a single instruction. The fast paths are the algebraic identities that the optimiser leaves for the back end because they depend on the target: multiplying by zero becomes `MOVE 0, R0`; by one, the operand itself; by negative one, a negation; and — the one that earns its keep most often — by a power of two becomes a left shift, so `x * 8` is `SHL R0, 3, R0` rather than a subroutine call. A handful of small non-power-of-two constants are open-coded as a short sequence of shifts and adds. Only when none of these applies does the generator fall back

IR operation	FRISC	Notes
add, sub	ADD, SUB	direct
and, or, xor	AND, OR, XOR	direct
shl, shr	SHL, SHR	direct
mul	SHL / adds / CALL F_MUL	fast path or helper
div, mod	CALL F_DIV / F_MOD	helper
cmp_*	CMP + JP_xx + MOVE	materialised (Section 8.4)
neg	SUB from 0	
not (logical)	CMP + JP_EQ + MOVE	like a comparison
trunc, zext	AND R0, 0FF, R0	narrow to a byte
sext	SHL then ASHR	sign-extend a byte
itof, ftoi	CALL F_I2F / F_F2I	helper
load, store	LOAD / STORE (or byte)	word or byte

Table 8.2: Instruction selection: each IR operation and the FRISC it lowers to. Most arithmetic and bitwise operations are a single instruction. Multiply, divide, modulo, and the two float conversions become helper calls because FRISC lacks the hardware. Comparisons and logical not must be *materialised* into a 0/1 value, which costs a small control-flow diamond.

to F_MUL, marking the helper as needed so that Chapter 9’s HelperEmitter knows to append its body.

The helper-call protocol itself is the same for every helper: arrange the two operands in R0 and R1, push them, CALL the helper, pop the arguments off, and move the result from R6 (where the helper left it) into R0. We will see the helper’s internals in the next chapter; here it is enough that the call follows the calling convention of Section 8.6 like any other.

8.3.3 Building large constants

The twenty-bit immediate field forces one more piece of selection. A constant that fits — and most do, since loop bounds and small offsets are tiny — goes straight into a MOVE. A constant that does not fit, such as a Q16.16 fixed-point literal or an address arithmetic that overflows the field, is assembled in two halves: load the high sixteen bits, shift them up, and OR in the low sixteen.

```

1      MOVE 0F, R0          ; high half of 0x000F4240 (1000000)
2      SHL R0, 10, R0      ; shift up by 16 bits
3      OR R0, 4240, R0     ; OR in the low half

```

It is a small detail, but it is the kind of detail Kernighan’s epigraph warns about: a low-level target makes us attend to the size of a number, something the IR never had to think about. The generator hides the attention inside ImmediateEmitter so that the rest of the lowerers can ask for “the constant *c* in register *R*” and not care whether it took one instruction or three.

8.4 Comparisons and control flow

A comparison is where the impedance mismatch between the IR and the machine is sharpest. The IR treats `t4 = cmp_le t3, #200` as an ordinary instruction that produces a Boolean value, to be stored in a slot like any other. FRISC has no instruction that produces the truth of a comparison as a value; it has `CMP`, which sets flags, and `JP_xx`, which consumes them by branching. To turn a comparison into a storable 0 or 1, the generator must *materialise* it: compare, branch over a `MOVE 0` to a `MOVE 1`, and let both arms rejoin. The result is a four-block control-flow diamond for what looked like a single IR instruction, drawn in Figure 8.3.

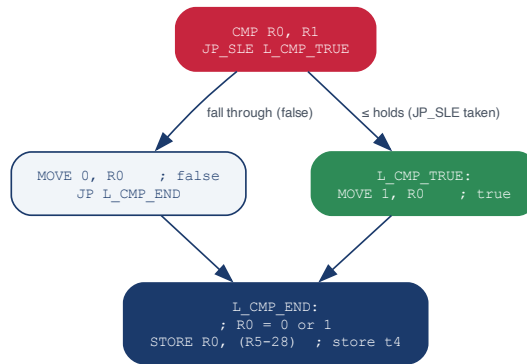


Figure 8.3: Lowering `t4 = cmp_le t3, #200` forces a control-flow diamond: `CMP/JP_SLE` in the entry block, a `MOVE 0` false arm and a `MOVE 1` true arm, and a join block where `R0` holds the materialised Boolean before it is stored into `t4`’s slot. A single IR comparison becomes four basic blocks of assembly.

Algorithm 8.3 is the materialisation. It begins with the familiar binary-operand dance — left to `R0`, push, right to `R1`, pop — because a comparison is, mechanically, a binary operation whose “result” happens to be a flag. The only new ingredient is the table that maps each IR comparison operator to its FRISC conditional jump, and the two fresh labels needed to name the true arm and the join point.

This is the part of code generation where a smarter back end would obviously do better, and it is worth naming the missed opportunity because the optimiser has already done most of the work to expose it. A great many comparisons exist only to be branched on immediately — `t4 = cmp_le t3, #200` followed at once by `br t4, L2, L4`. Materialising `t4` into a 0 or 1 and then immediately comparing that 0 or 1 against zero to decide the branch is pure waste: we computed a flag, threw it away by turning it into a number, and then recomputed a flag from the number. A back end that fused the comparison with the branch would emit a single `CMP/JP_SLE` straight to the target block. FRISCcc does not fuse, because

Algorithm 8.3: Materialising a comparison, from CompareLowerer. The operator-to-jump map sends `cmp_eq` to `JP_EQ`, `cmp_ne` to `JP_NE`, `cmp_lt` to `JP_SLT`, `cmp_le` to `JP_SLE`, `cmp_gt` to `JP_SGT`, and `cmp_ge` to `JP_SGE`.

Input: a comparison $t_d = \text{cmp}(a, b)$

Output: FRISC leaving 0 or 1 in `R0`

```

1 EVALVALUE(a, into R0); emit PUSH R0
2 EVALVALUE(b, into R0); emit MOVE R0, R1; emit POP R0
3  $\ell_{true} \leftarrow \text{NEWLABEL}(\text{L\_CMP\_TRUE})$ 
4  $\ell_{end} \leftarrow \text{NEWLABEL}(\text{L\_CMP\_END})$ 
5 emit CMP R0, R1
6 emit JUMP(cmp)  $\ell_{true}$                                 (e.g. JP_SLE for cmp_le)
7
8 emit MOVE 0, R0; emit JP  $\ell_{end}$                         (false arm)
9
10 emit label  $\ell_{true}$ ; emit MOVE 1, R0                    (true arm)
11
12 emit label  $\ell_{end}$ 
13 STORETEMP(d, from R0)
```

lowering each IR instruction in isolation is the simplicity it trades performance for; the diamond is the visible price of that isolation, and the peephole pass is not powerful enough to recover the fusion. We flag it here and return to it in the performance chapter.

8.4.1 Branches, jumps, and labels

Once a comparison has been materialised into a slot, branching on it is simple. The IR’s `br t, L_true, L_false` loads `t` into `R0`, compares it against zero, jumps to the true block if it is non-zero, and falls through to a jump to the false block. An unconditional `jmp L` is a bare `JP`. The destinations are FRISC labels, and the generator builds them with a small, total naming scheme so that no two labels in the program collide.

A function’s basic blocks are named by combining the function name with the IR block label: block `L0` of `main` becomes `L_MAIN_L0`, block `L11` becomes `L_MAIN_L11`, and a one-to-one map between IR labels and FRISC labels is built once per function so that branches resolve consistently. Labels the generator invents for its own purposes — the true and join arms of a comparison, the zero-initialisation loop in the prologue, the body of a byte-copy — are minted from a single global counter with a descriptive prefix, producing the `L_CMP_TRUE_3`, `L_ZERO_2`, and `L_MEMCPY_1` we see scattered through the listings. Function and global labels get the `F_` and `G_` prefixes we have already met: `main` becomes `F_MAIN`, the array `isPrime` becomes `G_ISPRIME`.

8.5 Memory: loads, stores, and addresses

The IR computes addresses with three instructions — `addr_of_symbol`, `addr_field`, and `addr_index` — and reads or writes through them with `load` and `store`. Lowering them is where the frame layout of [Section 8.2.1](#) and the data section of [Section 8.7](#) finally meet, because an address is just a number and the generator has to know which base to add the offset to.

`addr_of_symbol` produces the address of a named thing, and the base depends on where that thing lives. A global is its own label, so `addr_of_symbol global:isPrime` is simply `MOVE G_ISPRIME, R0`. A local lives in the current frame, below the frame pointer, so the generator computes `R5` minus the local's offset — by the same four-byte-plus-position reasoning as the temporary slots of [Section 8.2.1](#), the offset is four plus the local's position, putting count at `R5-0C`. A parameter lives *above* the frame pointer, in the caller's territory, so its address is `R5` plus a positive offset. Three symbol kinds, three bases: a global label, `R5 - offset`, and `R5 + offset`.

Reading and writing through an address is a `LOAD` or a `STORE`, with one wrinkle: FRISC distinguishes word access from byte access, and our IR carries the element type, so a char load becomes `LOADB` and a char store becomes `STOREB`. Because FRISC's `LOADB` zero-extends, a value read from a char slot is clamped to its low byte with `AND R0, 0FF, R0` when the type demands it. A word store reuses the push/pop idiom to protect the address while the value is computed: evaluate the address into `R0`, push it, evaluate the value into `R0`, pop the address into `R1`, and `STORE R0, (R1)`.

8.5.1 Array indexing and bounds checks

`addr_index` is the most elaborate address form, because a subscript `isPrime[p]` has to do real arithmetic — $base + index \times elementSize$ — and, unless the optimiser could prove the index in range, has to guard that arithmetic with a bounds check. [Figure 8.4](#) shows the whole computation for `isPrime[p]`, and [Algorithm 8.4](#) states it.

The bounds check deserves a word, because whether it appears at all is a small optimisation in itself. Emitting two comparisons and two conditional jumps on every array access would be safe but wasteful, so the generator runs a static analysis — `AddrIndexAnalyzer` — that asks, for each index, whether it can be proved at compile time to fall within the array. An access like `isPrime[0]` with a constant, in-range index needs no guard and gets none; an access with a computed index like `isPrime[p]`, whose value the generator cannot pin down, keeps both guards. When a guard fires it jumps to `L_BOUNDS_ERROR`, a single shared trap that halts the machine with an error code; we leave its body, and the question of what “halts with an error code” means on a simulator, to [Chapter 9](#). The point for instruction selection is only that a subscript is base-plus-scaled-index, optionally bracketed by two guards, and that the optimiser and a small analysis between them keep the guards off the paths that provably do not need them.

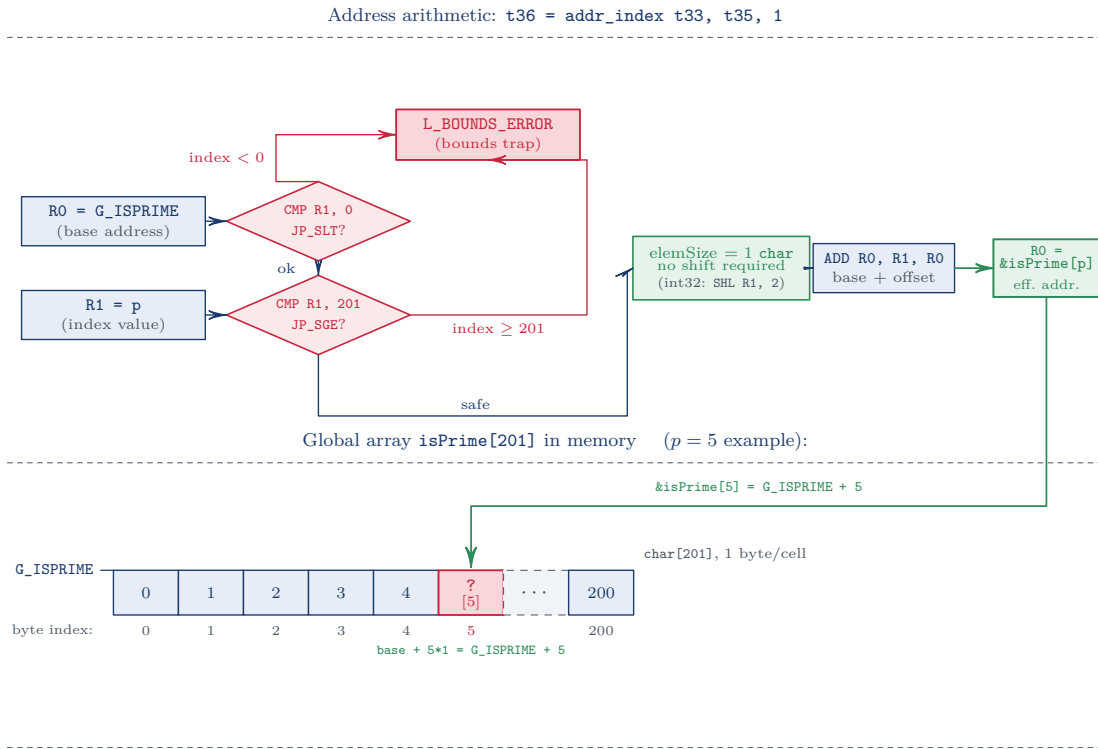


Figure 8.4: Lowering $t36 = \text{addr_index } t33, t35, 1$ — the access `isPrime[p]`. The base address of the array and the index are computed into `R0` and `R1`; two `CMP/jump` guards trap to `L_BOUNDS_ERROR` if the index is negative or at least the array size; the index is scaled by the element size (a no-op here, since `isPrime` is a `char` array with one-byte elements); and `ADD R0, R1, R0` forms the effective address. For a word array the scale step would be `SHL R1, 2, R1`.

Algorithm 8.4: Lowering `addr_index`, from `AddressLowerer`. The bounds check is emitted only for index temporaries that the `AddrIndexAnalyzer` could not prove in range; a power-of-two element size scales by a shift, and other sizes are open-coded.

Input: $t_d = \text{addr_index}(\text{base}, \text{index}, s)$ with element size s and array length n

Output: the effective address in `R0`

```

1 EVALVALUE(base, into R0); emit PUSH R0
2 EVALVALUE(index, into R0); emit MOVE R0, R1; emit POP R0
3 if d needs a bounds check then
4   | emit CMP R1, 0; emit JP_SLT L_BOUNDS_ERROR
5   | emit CMP R1, n; emit JP_SGE L_BOUNDS_ERROR
6 if  $s = 1$  then
7   | do nothing                                     (byte element: no scaling)
8 else if  $s = 2^k$  then
9   | emit SHL R1, k, R1
10 else open-code  $R1 \leftarrow R1 \times s$  with shifts and adds
11 emit ADD R0, R1, R0                             (base + scaled index)
12
13 STORETEMP(d, from R0)

```

8.6 The calling convention

The third question — how functions call each other — is answered by a protocol that every function obeys on both sides of every call. The protocol has three pieces: the entry stub that starts the program, the prologue and epilogue that bracket each function, and the argument-passing dance at each call site. Because all of FRISCCc’s functions agree on the protocol, a caller can hand control to a callee and get it back with the return value in `R6` and its own frame intact, without either side knowing anything about the other beyond its label and its arity.

The program begins not in `main` but in a three-instruction entry stub the generator emits first: set the stack pointer to the top of memory, call `main`, and halt.

```

1      MOVE 40000, R7      ; Initialize stack pointer (SP)
2      CALL F_MAIN        ; Call main
3      HALT               ; Program end

```

The stack pointer starts at 40,000 and grows downward from there, which is why every frame is carved out by *subtracting* from `R7`. The `HALT` after the `CALL` is the program’s only natural way to stop: when `main` returns, control falls back to the instruction after the `CALL`, which ends the run.

8.6.1 Prologue and epilogue

Each function brackets its body with a prologue that builds its frame and an epilogue that tears it down, and the two are mirror images. Algorithm 8.5 states both. The prologue saves the caller's frame pointer, adopts the stack pointer as its own frame pointer, reserves the frame computed in Algorithm 8.1, and — because the IR's semantics assume locals start at zero — clears the locals zone with a small loop. The epilogue releases the frame, restores the caller's frame pointer, and returns.

Algorithm 8.5: Function prologue and epilogue, from FunctionEmitter. The prologue saves the caller's frame pointer, establishes a new one, reserves the frame, and zeroes the locals; the epilogue undoes the frame and returns. Every return statement in the body jumps to the shared exit label — of the form `L_EXIT_F_f` with a per-program counter suffix, e.g. `L_EXIT_F_MAIN_1` — after placing its value in `R6`.

Input: a function f with frame size S and locals zone of w words

```

1 Prologue
2 emit label F_f
3 emit PUSH R5                                (save caller's frame pointer)
4
5 emit MOVE R7, R5                            (this frame's FP = current SP)
6
7 if  $S > 0$  then
8   emit SUB R7,  $S$ , R7                        (reserve the frame)
9
10  emit a loop L_ZERO that stores 0 to each of the  $w$  locals words
11 ... function body ...
12 Epilogue
13 emit label L_EXIT_F_f                      (with a per-program counter suffix)
14
15 if  $S > 0$  then emit ADD R7,  $S$ , R7
16 emit POP R5; emit RET

```

The prologue of `main` is exactly this template with the numbers filled in:

```

1  F_MAIN                                ; Function: main
2      PUSH R5                          ; Save old FP
3      MOVE R7, R5                      ; Set FP
4      SUB R7, 10C, R7                  ; Allocate locals/temps
5      MOVE 5, R1                       ; Zero local words
6      MOVE R5, R0                      ; Local zero base
7      SUB R0, 14, R0                   ; Local zero ptr
8      MOVE 0, R2                       ; Zero
9  L_ZERO_2

```

```

10      STORE R2, (R0)          ; Clear
11      ADD R0, 4, R0
12      SUB R1, 1, R1
13      JP_NE L_ZERO_2

```

The five words it zeroes (the count in `MOVE 5, R1`) are the twenty bytes of locals (`SUB R0, 14, R0`, since 14 is hexadecimal twenty); the temporaries are not cleared, because every temporary is written before it is read by construction of the IR. A `return count` in the body loads `count` into `R0`, moves it to the return register `R6`, and jumps to the shared exit label, where the epilogue releases the 268-byte frame, pops the saved frame pointer, and returns to the entry stub's `HALT`.

8.6.2 Passing arguments

A call has to get its arguments to the callee, and where they go depends on the callee's parameter layout, computed once for every function by `ParamLayoutBuilder`. The generator evaluates each argument in turn into `R0` and stages it in the argument-scratch zone of the current frame — the third zone of [Figure 8.2](#). When all arguments are staged, it reserves the callee's parameter area on the stack, copies the staged arguments into it (a word at a time for scalars, a byte-copy loop for an aggregate passed by value), and emits the `CALL`. After the call returns, it releases the parameter area and, if the callee returns a value, moves `R6` into `R0` so the result is where the next instruction expects it. `real_prime_sieve` has only the parameterless `main`, so its listings never exercise the user-function argument path (the `F_MUL` calls it does make follow the simpler helper protocol above), but the protocol is what makes any multi-function program — the case studies of [Chapter 13](#) — hang together.

A frugal trick: pointer scratch. One whole-program analysis in [Figure 8.1](#) exists purely to save stack space. A pointer local that is initialised once to the address of some object and never reassigned does not really need to live in the frame at all; its value is a constant for the function's lifetime. `PointerScratchCollector` finds such immutable pointer locals and backs each with a word of global storage — a label like `G_SCRATCH_main_buf` — that the prologue initialises once. It is a minor optimisation, invisible in `real_prime_sieve` because it has no pointer locals, but it is a nice illustration of how even a deliberately simple back end can afford one or two targeted cleverness when the analysis to justify them is cheap.

8.7 Globals and the data section

Everything so far has been code; a program also has data that outlives any single frame. Global variables and arrays are emitted into a data section after the last function, each

under its `G_` label, using three FRISC assembler directives: `DW` defines an initialised word, `DB` defines an initialised byte, and `'DS` reserves a block of uninitialised bytes. A scalar `int` global becomes a `DW` with its initial value; a `char` becomes a `DB`. An array's element type decides the directive — a word array is a `DW` list, a byte array a `DB` list — and an uninitialised array is reserved wholesale with `'DS`. Our anchor's lone global is exactly this last case:

```
1 | G_ISPRIME 'DS 201 ; uninitialized array
```

Two hundred one bytes of zeroed storage, one per element of `char isPrime[201]`, addressed from the code by the label `G_ISPRIME` we saw the index computation add its offset to. The generator tracks alignment as it lays the section out so that word data falls on word boundaries, and it appends any pointer-scratch words from the sidebar above at the end. The data section is the simplest part of the back end, and a good place to catch our breath before the last mechanism.

8.8 A second look: the peephole pass

Selecting instructions one IR operation at a time produces correct assembly and a predictable amount of litter. Evaluating an operand into `R0` and immediately moving it to `R0` leaves a `MOVE R0, R0`; a push immediately followed by its matching pop is a round trip to the stack that changed nothing; an `ADD R0, 0, R0` adds nothing; a `JP` to the very next line is a jump to where we were going anyway. None of these is a bug, and each is the natural seam between two independently-lowered pieces. Rather than complicate every lowerer to avoid them, FRISCcc cleans them up afterward, with a small peephole optimiser that scans the emitted instruction stream and rewrites local patterns.

Key insight

The back end gets its own tiny optimiser, for the same reason the front end did. The IR optimiser of [Chapter 7](#) could not see addressing modes or registers; the peephole pass cannot see across basic blocks or reason about values. Each cleans up the redundancy that its own upstream stage introduced by being deliberately literal. Splitting the work this way — a machine-independent optimiser on the IR, a machine-dependent peephole pass on the assembly — is the standard division of labour in real compilers, and it is why neither stage has to be clever about the other's concerns.

[Algorithm 8.6](#) lists the patterns. The pass runs them over the whole instruction list repeatedly until a full sweep changes nothing, because removing one instruction can expose another — a `POP` whose matching `PUSH` was deleted, a jump to a label that a deletion has just made the next line. Each rewrite only ever deletes instructions, and only when deletion is provably behaviour-preserving, so the fixed point is reached quickly and the result is never longer than the input.

Algorithm 8.6: The FRISC peephole optimiser, from `FriscPeepholeOptimizer`. Four local patterns — self-moves, push/pop round trips, identity arithmetic, and jumps to the next line — are deleted, and the pass iterates to a fixed point because each deletion can expose another.

Input: the emitted instruction list

Output: a shorter, equivalent list

```

1 repeat
2   for each instruction  $\iota$  (with one-line lookahead) do
3     if  $\iota$  is MOVE Rx, Rx then
4       | delete  $\iota$ 
5     else if  $\iota$  is PUSH Rx immediately followed by POP Rx then
6       | delete both
7     else if  $\iota$  is OP Rx, 0, Rx for an identity OP then
8       | delete  $\iota$ 
9     else if  $\iota$  is JP  $\ell$  and the next code line is label  $\ell$  then
10      | delete  $\iota$ 
11 until a full pass made no change

```

The peephole pass is the last thing to touch the code before it is written to `a.frisc`. It is modest by design — it does not fuse comparisons with branches, does not coalesce loads from the same slot, does not do any of the cross-instruction reasoning that the missed comparison-fusion of [Section 8.4](#) would require. Those are optimisations, and optimisations that need a value model live upstream on the IR. The peephole pass only removes instructions that are self-evidently inert, and that narrow remit is what keeps it, like the rest of the back end, small enough to trust.

8.9 Worked example: `real_prime_sieve`, all the way down

We have met every mechanism the generator uses; now we watch them work together on a single piece of the anchor. The inner loop of the sieve crosses off multiples of a prime: when `isPrime[p]` is set, it walks `j` from `p*p` upward in steps of `p`, clearing `isPrime[j]` each time. In optimised IR, the block that clears one multiple and advances `j` reads like this:

```

1  L11:
2      t33 = addr_of_symbol global:isPrime
3      t34 = addr_of_symbol local:j
4      t38 = addr_of_symbol local:p
5      t35 = load t34 : int32
6      t36 = addr_index t33, t35, 1
7      store t36, #0:int32 : char
8      t37 = load t34 : int32
9      t39 = load t38 : int32

```

```

10      t40 = add t37, t39 : int32
11      store t34, t40 : int32
12      jmp L10

```

Read against the chapter, every line of this IR now predicts its assembly. The three `addr_of_symbol` instructions are a global label and two FP-relative subtractions. The load `t34` is a word load through the address of `j`. The `addr_index t33, t35, 1` is the array-subscript pattern of [Algorithm 8.4](#) — base in `R0`, index in `R1`, two bounds-check guards because the index is computed, no scaling because the element is a byte, and an `ADD`. The `store t36, #0` is a byte store of zero through that address. The closing `add` and `store` advance `j` by `p`, and the `jmp` closes the loop. The store itself — the three address computations, the two bounds-check guards, and the byte write that clears `isPrime[j]` — is [Listing 8.1](#).

The listing is longer than its IR by exactly the factor this chapter has been preparing us for. A single C assignment becomes some two dozen instructions, and the full block — the store above plus the `add` and `store` that advance `j` — runs to roughly forty-five. The inflation is entirely accounted for by the two decisions we made up front: spilling every temporary turns each IR value into a `STORE` and a later `LOAD`, and lowering each instruction in isolation means the bounds-check guards and the materialised addresses cannot be shared even when, to a human eye, they obviously could. This is the code `FRISCCc` runs, and in [Chapter 10](#) we will run it and count exactly what those decisions cost in cycles. Before that, [Chapter 9](#) owes us the bodies of the helpers we have been calling and the trap we have been jumping to — the runtime that this generated code assumes is waiting for it.

Practice

Exercise 8.1 (Applied · ★★☆☆)

Lower a binary operation by hand. Using the pattern in [Algorithm 8.2](#), write out the FRISC instructions for the IR instruction `t7 = sub t2, t5 : int32`, assuming `t2` lives at slot `R5-10`, `t5` at `R5-14`, and `t7` at `R5-1C`. How many of your instructions are the actual subtraction, and how many are the cost of having no register allocator?

Exercise 8.2 (Applied · ★☆☆)

Which fast path fired? For each multiply, state which `BinaryLowerer` fast path applies and give the FRISC it produces, or say that it falls back to `F_MUL`: (a) `t1 = mul t0, #16`; (b) `t1 = mul t0, #1`; (c) `t1 = mul t0, #0`; (d) `t1 = mul t0, #7`; (e) `t1 = mul t0, t2` where neither operand is constant.

```

93  L_MAIN_L11
94      MOVE G_ISPRIME, R0      ; Address of global
95      STORE R0, (R5-9C)      ; Store temp t33
96      MOVE R5, R0
97      SUB R0, 10, R0          ; Local address
98      STORE R0, (R5-0A0)      ; Store temp t34
99      MOVE R5, R0
100     SUB R0, 8, R0           ; Local address
101     STORE R0, (R5-0B0)      ; Store temp t38
102     LOAD R0, (R5-0A0)       ; Load temp t34
103     LOAD R0, (R0)           ; Load word
104     STORE R0, (R5-0A4)      ; Store temp t35
105     LOAD R0, (R5-9C)        ; Load temp t33
106     PUSH R0                 ; Save base address
107     LOAD R0, (R5-0A4)       ; Load temp t35
108     MOVE R0, R1             ; Index
109     POP R0                  ; Restore base
110     CMP R1, 0               ; Bounds check
111     JP_SLT L_BOUNDS_ERROR
112     CMP R1, 0C9
113     JP_SGE L_BOUNDS_ERROR
114     ADD R0, R1, R0          ; Base + offset
115     STORE R0, (R5-0A8)      ; Store temp t36
116     LOAD R0, (R5-0A8)       ; Load temp t36
117     PUSH R0                 ; Save address
118     MOVE 0, R0
119     POP R1                  ; Restore address
120     STOREB R0, (R1)         ; Store byte

```

Listing 8.1: The array store `isPrime[j] = 0` from the inner sieve loop of `real_prime_sieve`, lowered to FRISC assembly (the head of block `L_MAIN_L11`). The three `addr_of_symbol` temporaries become a global label and two FP-relative addresses, stored to their slots; the subscript `isPrime[j]` expands into the base-plus-index computation with its two `JP_*L_BOUNDS_ERROR` guards; and the write is the `STOREB` of a freshly loaded zero. Note how every temporary makes the round trip through its stack slot — the visible cost of spilling everything.

Exercise 8.3 (Applied · ★★☆☆)

Materialise a comparison. Write the FRISC for `t9 = cmp_gt t8, #0 : bool` following [Algorithm 8.3](#), inventing the two labels you need. Then suppose the very next IR instruction is `br t9, L_pos, L_neg`. Sketch the single CMP/conditional-jump sequence a back end that fused comparison with branch would emit instead, and count the instructions saved.

Exercise 8.4 (Applied · ★★☆☆)

Size a frame. A function declares twelve bytes of locals and, after optimisation, uses temporaries `t0` through `t9` and makes one call passing two arguments. Using [Algorithm 8.1](#), compute the locals area, the temporary area, the argument-scratch area, and the total frame size. At what FP-relative offset does `t0` live?

Exercise 8.5 (Applied · ★★☆☆)

When does the guard disappear? `AddrIndexAnalyzer` elides bounds checks it can prove unnecessary. For each access into `int a[10]`, say whether both guards or no guards are emitted, and why (the analyser is all-or-nothing: it never emits exactly one of the two): (a) `a[0]`; (b) `a[i]` for a parameter `i`; (c) `a[k]` where the preceding IR is `k = #3`; (d) `a[n-1]` for an unknown `n`.

Project

Exercise 8.6 (Lab · ★★★)

Project: a code generator for tinyC. In the companion repository, the `ch08-codegen/` module contains a starter back end for tinyC — the int-only C subset with no arrays, structs, or floats. Your job is to turn tinyC’s optimised IR into FRISC. Because tinyC has no arrays, you may ignore `addr_index` and bounds checks entirely; because it has no floats, you need no `F_Fmul` or conversions; and because its only missing operations are multiply, divide, and modulo, your helper needs are `F_Mul`, `F_Div`, and `F_Mod` (whose bodies you will write in [Chapter 9](#)).

Implement, in order: the temporary-to-slot model and frame sizing ([Algorithm 8.1](#)); the binary-operation pattern with the multiply fast paths ([Algorithm 8.2](#)); comparison materialisation ([Algorithm 8.3](#)); the prologue and epilogue ([Algorithm 8.5](#)); and finally the four-pattern peephole pass ([Algorithm 8.6](#)). The mechanism is the same one FRISCcc uses; keeping each lowerer literal and leaning on the peephole pass afterward is the simplest path to passing the tests.

Your turn

Open the `ch08-codegen/starter/` module in the companion repository and implement `FriscEmitter.java` and `FunctionLowerer.java` following the algorithms above. Run `mvn test -pl ch08-codegen` from the companion root; the tests assemble your output with the bundled FRISC assembler and run it on the simulator, checking the return value of a dozen small programs — arithmetic, a factorial, a couple of loops. When they pass, you have a working back end: optimised IR in, runnable FRISC out. In [Chapter 9](#) you will give it the helpers it calls, and in [Chapter 10](#) you will watch it run.

Notes and further reading

The Kernighan epigraph captures the chapter’s recurring theme exactly: a low-level target makes the compiler attend to things — register pressure, immediate widths, the absence of a multiplier — that the IR was designed to ignore. The code generator is where that attention is paid.

The instruction selection we do is the simplest possible kind, a one-to-one macro expansion of each IR operation. The classical alternative treats selection as tree pattern matching with costs, and the foundational paper is Aho, Ganapathi, and Tjiang’s twig [1]; Fraser, Hanson, and Proebsting’s *iburg* [29] is the readable, practical descendant, and bottom-up rewrite systems of that lineage are the standard technique in production back ends. Appel’s *Modern Compiler Implementation* [7] gives a clear textbook treatment of selection, the calling convention, and the frame layout, and was a close reference while writing this chapter.

Register allocation is the back-end topic we most conspicuously skip, and the two papers in the sidebar are the places to start: Chaitin’s graph-colouring formulation [15] and Poletto and Sarkar’s linear scan [71], the latter being the allocator of choice when compilation speed matters, as in just-in-time compilers. Briggs, Cooper, and Torczon’s improvements to graph colouring [13] are the refinements most real colouring allocators actually implement. Cooper and Torczon’s *Engineering a Compiler* [18] devotes its closing chapters to selection, allocation, and instruction scheduling, and is the modern reference whose taste in back-end design most resembles ours — minus, of course, the allocator we declined to build.

Peephole optimisation goes back to McKeeman’s 1965 note [59]; the idea of cleaning up a naive instruction stream with a small fixed set of local rewrites is almost as old as code generation itself, and remains a standard final stage in real compilers. Davidson and Fraser [22] showed how to make such a pass retargetable by driving the rewrites from a machine description rather than hand-coding them, the lineage our four hand-written patterns are a deliberately small instance of. The division of labour we lean on — a machine-independent optimiser on the IR, a machine-dependent peephole pass on the assembly — is described in the Dragon Book [2] and is the arrangement nearly every production compiler adopts.

The calling convention FRISCcc uses is a deliberately simplified relative of the real-world application binary interfaces, the most widely documented of which is the System V AMD64 ABI [58]; reading even its first few pages is a quick way to appreciate how much complexity — register save masks, red zones, variadic argument areas, stack alignment guarantees — a teaching convention is free to omit. The semantic-preservation argument that the generated FRISC faithfully implements the IR it was lowered from is, like the optimiser’s, deferred to [Appendix G](#); CompCert [56] remains the reference for what a fully verified version of this chapter’s correctness claim looks like.

Chapter 9

Runtime: helpers, traps, and frame layout

*“When in doubt, use brute force.”
— Ken Thompson*

When we left the code generator in [Chapter 8](#), it ended on a small dishonesty, and we promised to make it good. Whenever the code generator met an IR `mul` it could not fold away, it emitted three words — arrange the operands, `CALL F_MUL`, take the result from `R6` — and moved on as though `F_MUL` were a primitive of the machine. It is not. FRISC has no multiply instruction, no divide, no modulo, nothing at all that understands a fraction, and no notion that an array index might point outside its array. `F_MUL` is a subroutine that FRISCcc writes, in FRISC assembly, and staples to the end of every program that multiplies two integers. So are `F_DIV`, `F_MOD`, the four fixed-point float routines, and the bounds-check trap. Together they are the program’s *runtime*: the code a compiled program carries that its author never wrote but cannot run without.

Pull `F_MUL` apart and you find no primitive at all, only a loop of shifts and adds — and this chapter pulls every such label apart. It is the shortest distance in the book between an idea and a consequence, because every gap between what our C subset promises and what FRISC delivers has to be closed *somewhere*, and the runtime is where. The language promises multiplication; the machine offers shifts and adds, so the runtime multiplies the way we were taught to in primary school, one partial product at a time. The language promises division; the machine offers subtraction and comparison, so the runtime divides bit by bit, the way long division works on paper. The language admits `float`; the machine has no floating-point unit, so the runtime represents a fraction as an ordinary integer scaled by a fixed power of two and does arithmetic on the scaled integers. And the language — ours, at least — promises that an out-of-bounds array access stops the program rather than corrupting it, so the runtime checks every index and traps when one is wrong.

None of this is glamorous, and that is rather the point. The runtime is the layer where high-level guarantees are paid for in the machine’s own coin, and the bill always comes due. A single `*` in the source becomes a loop of thirty-odd instructions; a single `a[i]` grows two comparisons and a conditional jump. Watching the runtime is the most direct way to feel the weight of an abstraction, and it explains, after the fact, why [Chapter 7](#) worked so hard

to delete multiplies the program did not really need and why [Chapter 14](#) will find so many cycles hiding in so few source characters.

We proceed from the outside in. First we fix what a runtime is and how FRISCCc decides which pieces of it a given program needs. Then we study the call handshake — the contract every routine, helper or not, obeys when it hands control to another — because the helpers are ordinary functions and obey it too. With the contract in hand we open the integer helpers, then the fixed-point float helpers, then the bounds-check trap. We close, as always, by watching the whole runtime assemble itself around `real_prime_sieve` and handing the result to the simulator of [Chapter 10](#).

9.1 What a runtime is

The word *runtime* is overloaded, so let us pin down the sense we mean. We do not mean a virtual machine, or a garbage collector, or a scheduler — the heavyweight runtimes of Java or Go. We mean the much older and humbler thing: a handful of subroutines, written once in the target’s own assembly, that implement operations the source language assumes but the hardware does not provide. Every C compiler has one. On a machine without hardware division, even a single `a / b` in your program links in a routine, conventionally named something like `__divsi3`, that the compiler vendor wrote in assembly. FRISCCc’s runtime is exactly this kind of library, with the pedagogical virtue that we can read all of it in an afternoon.

Where does it live? Recall from [Chapter 8](#) that the `ProgramEmitter` lays a program down in a fixed order: first the entry stub that boots the machine and calls `main`, then every user function in source order, then the helper routines, and finally the data section with the globals. The runtime is that second-to-last band — a contiguous run of routines, each behind a label beginning `F_` (for the arithmetic helpers) or `L_` (for the bounds trap), emitted after the last user function and before the first global. It is part of the same flat assembly file as everything else; there is no linker, no separate library, no dynamic loading. The runtime is simply more code, appended.

Key insight

The runtime is the part of the program nobody wrote but everybody needs.

It is not a service the operating system provides — there is no operating system. It is not magic the hardware performs — the hardware does shifts, adds, and conditional jumps, and nothing more. It is library code, in the target’s own assembly, that stands in for the instructions the machine lacks. Every abstraction the source language offers above the bare ISA is cashed out here, in full, in the machine’s own coin.

9.1.1 Pay only for what you use

A program that never multiplies should not carry a multiply routine. FRISCcc honours this by emitting helpers *on demand*: the generator records, as it lowers each function, which operations it actually produced a call for, and the helper library consults those records when it decides what to append. Mechanically, the emitter exposes a family of boolean flags — `needsMul`, `needsDiv`, `needsMod`, `needsFmul`, `needsFdiv`, `needsI2f`, `needsF2i`, `needsBoundsCheck` — each set the first time the generator emits a call to the corresponding routine. The `HelperLibrary` then walks the flags in a fixed order and emits each routine only if its flag is set. A program that does integer arithmetic on arrays, like `real_prime_sieve`, pulls in `F_MUL` and the bounds trap and nothing else; a program that divides floats pulls in the whole fixed-point apparatus and, as we will see, the integer division routine underneath it.

Figure 9.1 traces the dependency from a source-level feature, through the flag it sets, to the routine that ends up in the assembly. It is worth reading left to right once: the language feature on the left is a promise, the flag in the middle is the generator noticing the promise was used, and the routine on the right is the promise being kept.

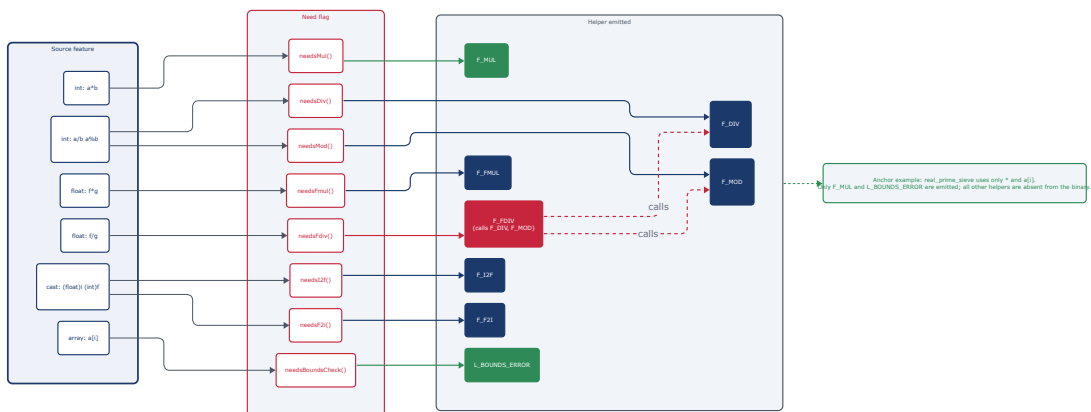


Figure 9.1: On-demand emission of runtime helpers. Each source-language feature that the hardware cannot perform sets a needs flag the first time the code generator lowers it; the `HelperLibrary` emits a routine only when its flag is set. Float division is the interesting case: its routine, `F_FDIV`, internally calls the integer `F_DIV` and `F_MOD`, so using one float divide silently pulls in two integer helpers. The anchor program `real_prime_sieve` sets only `needsMul` and `needsBoundsCheck`, so its runtime is just `F_MUL` and the `L_BOUNDS_ERROR` trap (highlighted).

The dependency edge from `F_FDIV` down to `F_DIV` and `F_MOD` is the one detail the flag machinery does not capture by itself, and it matters. A float divide is implemented in terms of an integer divide and an integer modulo, so the routines that emit `F_FDIV` also set `needsDiv` and `needsMod` before they finish, guaranteeing the integer helpers are present by the time the float helper calls them. The lesson generalises: a runtime is a small program

in its own right, with its own internal call graph, and “which helpers does this program need?” is a reachability question over that graph, not a flat checklist.

Hosted versus freestanding. The C standard distinguishes a *hosted* implementation, which runs under an operating system and may assume the full standard library, from a *freestanding* one, which may not. FRISCCc is about as freestanding as it gets: there is no `libc`, no `printf`, no heap, no file system, and the program-startup code is three instructions rather than the few hundred that a Unix C runtime’s `crt0` executes before it ever reaches `main`. What we call the runtime here is only the arithmetic-and-safety sliver that even a freestanding implementation cannot do without. It is a useful reminder of how much of a “normal” C program’s behaviour lives in code the compiler did not generate from your source at all.

9.2 The call handshake

The helpers are not special. `F_MUL` is a function: it is entered with `CALL`, it reads its arguments from the stack, it leaves its result in `R6`, and it returns with `RET`. Everything the code generator does to call a user function it does, identically, to call a helper. So before we can read a single helper we have to be precise about the contract every routine obeys — the *calling convention* — and in particular about the small region of the stack where a caller and a callee meet. Chapter 8 sized the frame; here we walk the handshake that frame exists to support.

9.2.1 The linkage region

When one routine calls another, control crosses a boundary, and a few words of stack straddle it. Reading the prologue of any FRISCCc function from the inside out tells the whole story. A caller wishing to invoke `f(a, b)` pushes the arguments onto the stack, then executes `CALL`, which pushes the return address and jumps to `f`. The first thing `f` does is save the caller’s frame pointer and claim the stack pointer as its own:

```

1      PUSH R5                ; save the caller's frame pointer
2      MOVE R7, R5           ; this frame's FP = current SP
```

After these two instructions, `R5` points at the saved old frame pointer, and a fixed staircase of known offsets climbs above it: at `(R5+0)` the saved `R5`, at `(R5+4)` the return address `CALL` pushed, and at `(R5+8)`, `(R5+12)`, ... the arguments the caller pushed before that. This is why every helper in this chapter reads its first argument from `(R5+8)` and its second from `(R5+0C)` — `0C` being hexadecimal twelve, in the leading-zero convention Chapter 8 introduced. The region from the arguments down through the saved frame pointer is the *linkage region*: the

handful of words that two frames share, and the only channel through which a caller speaks to a callee.

Figure 9.2 draws the linkage region in place, bridging the caller’s frame above and the callee’s freshly allocated frame below. It is the same activation record Chapter 8 sized, seen now from the callee’s point of view — the view a helper has of the world when it wakes up.

9.2.2 Prologue and epilogue

Saving R5 is only the first half of taking over a frame. A function with locals or temporaries must also reserve room for them and — because our semantics give every local a defined initial value of zero — clear that room before the body runs. The full prologue and its mirror-image epilogue are Algorithm 9.1. They are the discipline that makes the linkage region trustworthy: every routine leaves the stack pointer, the frame pointer, and the return path exactly as it found them, so that a caller can issue a CALL and assume, on return, that nothing it cared about has moved.

Two details of Algorithm 9.1 repay attention. First, the epilogue restores the frame by *adding back* exactly the constant the prologue subtracted, rather than by reloading a saved stack pointer; this works precisely because the body is disciplined never to leave the stack unbalanced across its own span, an invariant the lowering walker of Chapter 6 maintains by construction. Second, only the *locals* zone is zeroed, not the temporaries: temporaries are always written before they are read — that is part of what well-formed IR means — so spending cycles to clear them would be waste. The helpers, having no declared locals at all, skip the zeroing loop entirely; their prologue is just the two-instruction PUSH/MOVE pair we read above.

9.2.3 The handshake, end to end

Putting the two sides together, a call is a short, strict dance. Figure 9.3 follows it for the multiply that `real_prime_sieve` performs when it evaluates `p * p`. The caller stages the two arguments, lowers the stack pointer to open an argument area, copies the arguments in, and issues CALL F_MUL. Control crosses into the helper, which runs the prologue, reads the arguments out of the linkage region, computes, leaves its answer in R6, and returns. Back on the caller’s side, the stack pointer is raised to discard the arguments — the caller cleans its own argument area, the convention Chapter 8 settled on — and the result is moved out of R6 into R0, ready to be stored into whatever temporary the IR named.

There is one routine that nobody calls in the ordinary way, and it is worth meeting now because it anchors the whole call chain: the entry stub. A FRISC machine starts executing at address zero, so the very first instructions FRISCCc emits initialise the world and hand control to `main`:

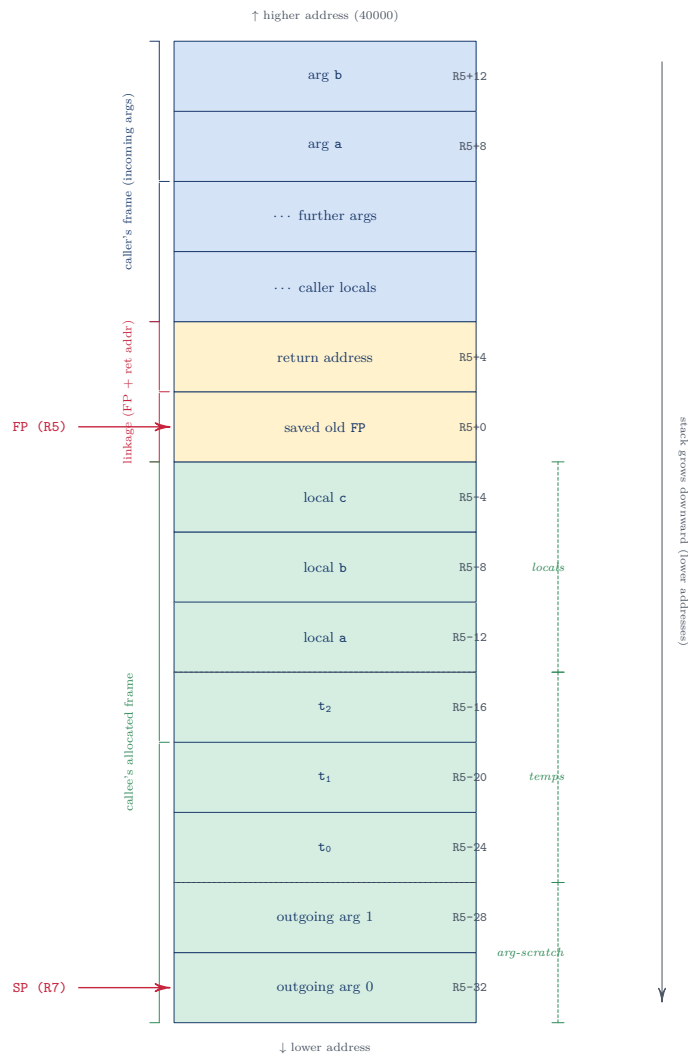


Figure 9.2: The stack at the moment a callee has just run its two-instruction prologue. The *linkage region* (centre) is shared: the caller pushed the arguments and CALL pushed the return address, then the callee saved the old frame pointer. R5 now points at that saved word, so the callee reads its arguments at the fixed offsets ($R5+8$) and ($R5+0C$) and its locals and temporaries at negative offsets below. R7 sits at the bottom after the prologue's SUB. The stack grows downward, toward lower addresses, from its initial value 40000.

Algorithm 9.1: The prologue and epilogue emitted by `FunctionEmitter` for every routine, helpers included. The prologue saves the caller’s frame pointer, adopts the current stack pointer as its own, reserves the frame, and zeroes the declared-locals zone; the epilogue undoes each step in reverse.

Input: a function with frame size S bytes and a locals zone of L bytes (a multiple of four)

Output: emitted prologue and epilogue assembly

```

1 Prologue:
2 EMIT PUSH R5                                (save caller’s frame pointer)
3
4 EMIT MOVE R7, R5                            (adopt SP as this frame’s FP)
5
6 if  $S > 0$  then
7   EMIT SUB R7, S, R7                        (reserve locals, temps, arg scratch)
8
9   if  $L > 0$  then
10     $p \leftarrow R5 - L$ ; emit a loop storing 0 to  $[p]$ ,  $[p+4]$ , ... for  $L/4$  words (zero the
11    locals zone)
12 ...body ...
13 Epilogue:
14 if  $S > 0$  then
15   EMIT ADD R7, S, R7                        (release the frame)
16
17 EMIT POP R5                                (restore caller’s frame pointer)
18
19 EMIT RET                                    (pop return address into PC)
20

```

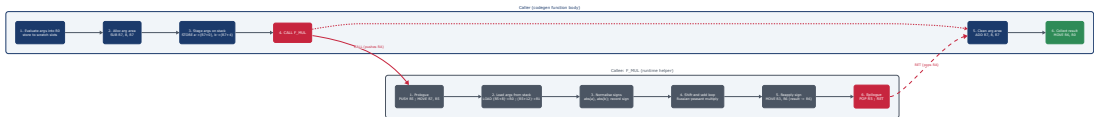


Figure 9.3: The call handshake for an integer multiply. The caller (left) stages the arguments, opens an argument area on the stack, and issues `CALL F_MUL`; the control transfer (red) crosses into the helper (right), which runs its prologue, reads the arguments from the linkage region at $(R5+8)$ and $(R5+0C)$, computes by shift-and-add, places the result in the return register $R6$, and returns. The caller then discards the arguments and moves $R6$ into $R0$. $R6$ is the return-value register; the caller cleans its own arguments.

```

1      MOVE 40000, R7      ; initialise the stack pointer
2      CALL F_MAIN        ; call main
3      HALT               ; main returned; stop the machine

```

The stack pointer is set to 40000, the top of the region the stack is allowed to grow down through, and then `main` is called like any other function. When `main` returns, there is no caller above it to return to, so the stub simply halts. Everything in between — every user function, every helper — is reached, directly or transitively, from that single `CALL F_MAIN`.

9.3 Arithmetic the machine refuses

Now we can read the helpers themselves, starting with the three the integer ALU forces us to write: multiply, divide, and modulo. All three follow the same overall plan. Reduce the problem to non-negative operands by recording the sign separately; run a bit-at-a-time loop that uses only the operations FRISC *does* have — shifts, adds, subtracts, comparisons, and conditional jumps; then reapply the sign to the result. The loops are textbook, in the most literal sense: they are the grade-school algorithms for multiplication and long division, transcribed for binary.

9.3.1 Multiplication by shift and add

To multiply a by b without a multiplier, observe that b is a sum of powers of two — its binary digits — and that multiplying a by a power of two is a left shift. So $a \cdot b$ is the sum of left-shifted copies of a , one for each 1-bit of b . This is precisely how we multiply by hand, except that each “digit” of the multiplier is 0 or 1, which removes the multiplication-table step entirely: we either add the shifted multiplicand or we do not. [Algorithm 9.2](#) is the routine, sign handling and all.

The assembly FRISCcc actually emits for `F_MUL` is a faithful transcription of [Algorithm 9.2](#), and reading it cements how little the machine is really doing per step. [Listing 4](#) is the routine exactly as it appears in the compiled `real_prime_sieve`. The sign of each operand is tested with `CMP` against zero and negated with a `SUB` from zero (FRISC has no unary negate); the parity test `b & 1` is an `AND` with the immediate 1; the shifts are `SHL` and `SHR` by one. The whole multiply is thirty-odd instructions and a loop that turns once per bit of the multiplier.

9.3.2 Division and modulo, bit by bit

Division is harder, and the routine shows it. The plan is *restoring division*: the same long division you do on paper, where at each step you bring down the next digit, ask whether the divisor fits, write a quotient digit accordingly, and subtract if it did. In binary the

```

1  F_MUL                                ; int32 multiplication
2      PUSH R5
3      MOVE R7, R5
4      LOAD R0, (R5+8)                  ; a
5      LOAD R1, (R5+0C)                 ; b
6      MOVE 0, R4                       ; zero
7      MOVE 0, R2                       ; sign
8      CMP R0, 0
9      JP_SGE L_MUL_A_POS_1
10     SUB R4, R0, R0                   ; a = 0 - a
11     XOR R2, 1, R2                   ; flip sign
12 L_MUL_A_POS_1
13     CMP R1, 0
14     JP_SGE L_MUL_B_POS_2
15     SUB R4, R1, R1                   ; b = 0 - b
16     XOR R2, 1, R2                   ; flip sign
17 L_MUL_B_POS_2
18     MOVE 0, R3                       ; result
19 L_MUL_LOOP_5
20     CMP R1, 0
21     JP_EQ L_MUL_DONE_3              ; no bits left
22     AND R1, 1, R4
23     CMP R4, 0
24     JP_EQ L_MUL_SKIP_4              ; this bit is 0
25     ADD R3, R0, R3                  ; result += a
26 L_MUL_SKIP_4
27     SHL R0, 1, R0                   ; a <= 1
28     SHR R1, 1, R1                   ; b >= 1
29     JP L_MUL_LOOP_5
30 L_MUL_DONE_3
31     CMP R2, 0
32     JP_EQ L_MUL_SIGN_DONE_6
33     MOVE 0, R4
34     SUB R4, R3, R3                   ; result = -result
35 L_MUL_SIGN_DONE_6
36     MOVE R3, R6                       ; return value
37     POP R5
38     RET

```

Listing 4: The F_MUL routine exactly as FRISCcc emits it for `real_prime_sieve`. The helper-local labels (L_MUL_*) are numbered by a deterministic counter so that two helpers never collide. This is the price of one source-level *.

Algorithm 9.2: Signed integer multiplication by shift-and-add. The operands are made non-negative and their combined sign is recorded; the loop adds a left-shifted copy of a for each set bit of b ; the sign is reapplied at the end. Every step uses only operations the FRISC ALU provides.

Input: integers a, b read from (R5+8), (R5+0C)

Output: $a \cdot b$, returned in R6

```

1   $sign \leftarrow 0$ 
2  if  $a < 0$  then  $a \leftarrow -a$ ;  $sign \leftarrow sign \oplus 1$ 
3  if  $b < 0$  then  $b \leftarrow -b$ ;  $sign \leftarrow sign \oplus 1$ 
4   $result \leftarrow 0$ 
5  while  $b \neq 0$  do
6      if  $b \& 1 = 1$  then
7           $result \leftarrow result + a$                                 (this bit of  $b$  is set)
8       $a \leftarrow a \ll 1$                                           (next power of two)
9       $b \leftarrow b \gg 1$                                           (consume the low bit)
10
11
12
13 if  $sign \neq 0$  then  $result \leftarrow -result$ 
14 return  $result$ 

```

“does it fit?” question has a yes/no answer and the quotient digit is one bit, so the routine shifts the dividend left into a running remainder one bit at a time, compares the remainder against the divisor, and on success subtracts and sets the low quotient bit. Thirty-two iterations later, every bit of the dividend has passed through and the quotient is complete. Algorithm 9.3 states the core loop, writing $MSB(n)$ for the most-significant (top) bit of n .

The real `F_DIV` wraps this loop in guards that the prose algorithm glosses over but that correctness demands, and they are instructive because each one is a place where the mathematics and the machine’s fixed-width arithmetic disagree. *Division by zero* cannot trap usefully here — there is no exception mechanism — so the routine returns zero, a defined-if-arbitrary result that keeps the program running. *The value INT_MIN divided by -1* is the one case where the true quotient, 2^{31} , does not fit in a signed 32-bit word; the routine detects it and returns `INT_MIN` itself, matching the wrap-around the rest of our two’s-complement arithmetic exhibits. And *division by -1* in general is special-cased to a simple negation, both as a speed-up and to route the `INT_MIN` case cleanly. `F_MOD` is the same routine with two differences: it returns the remainder rather than the quotient, and the sign it reapplies is the sign of the *dividend*, because in C the result of `%` takes the sign of the left operand.

Algorithm 9.3: The core of restoring division, shared by `F_DIV` (which returns q) and `F_MOD` (which returns r). The top bit of the dividend is shifted into the remainder each iteration; if the remainder reaches the divisor, it is reduced and a quotient bit is set. FRISC’s carry flag, set by `SHL`, is how the implementation reads `MSB( $n$ )`.

Input: non-negative dividend n , positive divisor d (after sign extraction)

Output: quotient q with remainder r , so that $n = qd + r$

```

1  $r \leftarrow 0$ ;  $q \leftarrow 0$ ;  $i \leftarrow 32$ 
2 while  $i > 0$  do
3    $r \leftarrow (r \ll 1) \mid \text{MSB}(n)$  (shift top bit of  $n$  into  $r$ )
4
5    $n \leftarrow n \ll 1$ 
6    $q \leftarrow q \ll 1$ 
7   if  $r \geq d$  then
8      $r \leftarrow r - d$ ;  $q \leftarrow q \mid 1$  (divisor fits: set quotient bit)
9
10   $i \leftarrow i - 1$ 
11 return  $(q, r)$ 
```

Key insight

A multiply is a loop; a divide is a longer one. On FRISC, `F_MUL` turns its loop once per bit of the multiplier and `F_DIV` turns its loop a full thirty-two times regardless of the operands. An integer multiply therefore costs tens of cycles and a divide costs more — against a single cycle for an `ADD`. This is the cycle-level reason the optimiser of Chapter 7 fought so hard to replace multiplies by shifts, hoist invariant multiplies out of loops, and fold constant arithmetic at compile time. Every helper call the optimiser deletes is a few dozen cycles that never run.

A small but real subtlety of reading these listings is that FRISC immediates are hexadecimal. The bit counter that `F_DIV` initialises with `MOVE 20, R4` is loading thirty-two, not twenty, because `20` is hex; the loop that decrements it to zero is the thirty-two iterations of Algorithm 9.3. Keeping the base in mind is the difference between a listing that looks wrong and one that reads correctly.

9.4 Fractions without an FPU

FRISC has no floating-point unit and no floating-point instructions, yet our C subset admits float. Something has to give, and what gives is the representation: FRISCcc does not implement IEEE-754 at all. Instead it represents a float as an ordinary 32-bit signed integer interpreted as a fixed multiple of 2^{-16} — the format called *Q16.16*. The high sixteen

bits are the integer part, the low sixteen are the fraction, and the value of the word w is simply $w/2^{16}$. Figure 9.4 lays out the bit field and works one example.

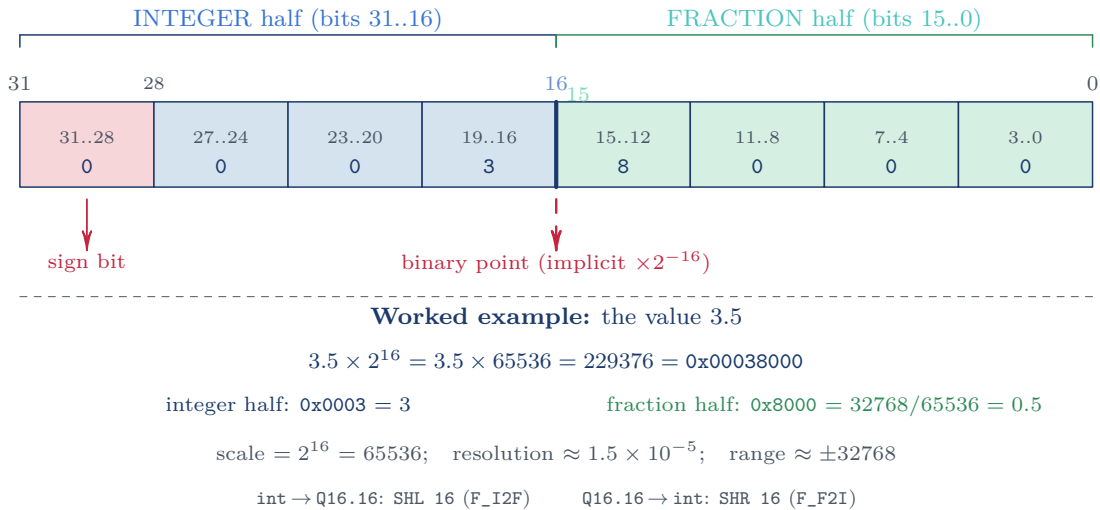


Figure 9.4: The Q16.16 representation of float. A 32-bit word is read as a two’s-complement integer scaled by 2^{-16} : the high sixteen bits are the integer part, the low sixteen the fraction, with an implicit binary point between them. The value 3.5 is stored as $3.5 \times 2^{16} = 229376 = 0x00038000$, whose integer half is 3 and whose fraction half, $0x8000$, is exactly one half. Conversions are pure shifts: scaling up by 2^{16} to enter the format, down by 2^{16} to leave it.

The choice trades range and precision for sheer simplicity. A Q16.16 number can represent magnitudes up to about ± 32768 with a resolution near 1.5×10^{-5} , far less than an IEEE float’s dynamic range, but it has a decisive advantage on a machine like ours: addition and subtraction of Q16.16 values are just integer addition and subtraction, because scaling distributes over them. Only multiplication, division, and the conversions need help, and the help is small.

9.4.1 Conversions are shifts

Entering and leaving the format is the easy part. To turn an integer i into Q16.16 we multiply by the scale 2^{16} , which is a left shift by sixteen; to turn a Q16.16 value back into an integer we divide by the scale, a right shift by sixteen that discards the fraction (an arithmetic shift, so it truncates toward zero for non-negative values and toward negative infinity for negative ones). The two helpers are correspondingly tiny — prologue, one shift, move to R6, epilogue:

```

1  F_I2F                                ; int32 to Q16.16
2      PUSH R5
3      MOVE R7, R5
4      LOAD R0, (R5+8)
5      SHL R0, 10, R0                    ; <=< 16 (10 is hex sixteen)
6      MOVE R0, R6
7      POP R5
8      RET

```

F_F2I is identical with SHR in place of SHL. Note again the hexadecimal: SHL R0, 10, R0 shifts by sixteen, the width of the fraction field, not by ten.

9.4.2 Fixed-point multiply and divide

Multiplication needs a moment's thought about scale. If a and b are the stored integers for two Q16.16 values, their true product as fixed-point numbers is $(a/2^{16})(b/2^{16}) = (ab)/2^{32}$, so the stored result we want is $ab/2^{16}$. But ab is a full 64-bit product, and the top of it does not fit in one register. F_FML therefore accumulates the product across *two* registers — a low word and a high word — using a double-width shift-and-add loop, and then extracts the middle 32 bits by shifting the low word right by sixteen and the high word left by sixteen and combining them. That “shift the 64-bit product right by 16” is the fixed-point correction; everything else is the integer multiply of [Algorithm 9.2](#) widened to two registers. [Algorithm 9.4](#) sketches it.

Division closes the circle by reusing the integer helpers. To compute a/b as fixed-point, F_FDIV first divides the stored integers as ordinary integers — one CALL F_DIV — to get the integer part of the quotient, then takes the remainder with one CALL F_MOD, and then develops sixteen fractional bits by the familiar shift-compare-subtract loop, each iteration doubling the remainder and asking whether the divisor now fits. This is why the dependency edge in [Figure 9.1](#) runs from F_FDIV down to F_DIV and F_MOD: a single float divide in your source pulls all three routines into the program. The fixed-point runtime is not flat; it is a little tower, with the integer division routine holding up the float one.

Why not IEEE-754? A standards-compliant single-precision float would need software routines to unpack a sign, an eight-bit exponent, and a 23-bit mantissa; align operands; round to nearest-even; and repack — hundreds of instructions per operation, and a great deal of subtlety about infinities, NaNs, and subnormals [32]. Fixed-point throws all of that away. There is no exponent to manage, so addition is free; there is no rounding mode to honour, so the routines are short; and the representation is just an integer, so the same ALU serves both int and float. The cost is a fixed, unforgiving range. For a teaching compiler whose goal is to show the whole path from source to silicon, Q16.16 is the honest choice: every float operation is something a reader can

Algorithm 9.4: Q16.16 multiplication. The integer product of the two stored values is a 64-bit quantity accumulated across a low and a high word using ADD/ADC (add-with-carry); the stored fixed-point result is that 64-bit product shifted right by sixteen, recovered by combining the two words. The ADC instruction and the carry flag are what make a double-width add possible on a 32-bit machine.

Input: stored Q16.16 integers a , b from (R5+8), (R5+0C)

Output: stored Q16.16 product $ab/2^{16}$, in R6

```

1 extract and record the combined sign; take  $|a|$ ,  $|b|$ 
2  $(lo, hi) \leftarrow (0, 0)$  (the 64-bit accumulator)
3
4 while  $b \neq 0$  do
5   if low bit of  $b$  is set then
6      $\lfloor$  add  $a$  into  $lo$ , carrying into  $hi$  (ADD then ADC)
7     shift the (widening) multiplicand  $a$  left by one across both words
8      $b \leftarrow b \gg 1$ 
9  $result \leftarrow (lo \gg 16) \mid (hi \ll 16)$  (divide the 64-bit product by  $2^{16}$ )
10
11 if sign set then  $result \leftarrow -result$ 
12 return  $result$ 
```

trace by hand.

9.5 Traps and the unhappy path

The helpers so far compute values. The last piece of the runtime does the opposite: it refuses to compute, deliberately, when continuing would be unsafe. Our C subset gives arrays a defined size, and indexing outside that size is not allowed to silently read or write a neighbouring variable. The code generator enforces this by emitting, at *every* array index, a pair of comparisons that check the index against the array's bounds, and a single shared trap that stops the machine if either check fails.

Reading the check in place makes its shape clear. Listing 5 is the indexing of `isPrime[i]` in `real_prime_sieve`, lifted straight from the compiled output. The index has already been computed into R1; the two CMP/JP pairs verify $0 \leq R1$ and $R1 < 201$ — the array `isPrime` has 201 entries, and 201 is 0C9 in hex — before the index is added to the base address. If either bound is violated, control jumps to `L_BOUNDS_ERROR`.

The trap itself is the smallest routine in the runtime, and it is emitted once no matter how many checks jump to it:


```

1      MOVE R0, R1          ; index into R1
2      POP R0               ; restore base
3      CMP R1, 0            ; bounds check: lower
4      JP_SLT L_BOUNDS_ERROR ; index < 0 -> trap
5      CMP R1, 0C9          ; bounds check: upper (0C9 = 201)
6      JP_SGE L_BOUNDS_ERROR ; index >= length -> trap
7      ADD R0, R1, R0       ; base + offset, now known safe

```

Listing 5: The bounds check FRISCcc inserts at the indexing of `isPrime[i]`. The lower-bound test rejects negative indices; the upper-bound test rejects indices at or beyond the array length. Only if both pass is the offset added to the base address. Every array access in the program carries its own copy of this guard.

```

1      L_BOUNDS_ERROR      ; array bounds error
2      MOVE -6, R6         ; error code into the result register
3      HALT                ; stop the machine

```

It loads a distinguished error code, -6, into the return register and halts. There is no operating system to raise a signal to, no console to print a diagnostic on, and no exception to unwind; HALT is the only “stop everything” the machine has, and a recognisable code in R6 is the only message the runtime can leave behind. A harness that runs the program and inspects the final register file can tell a bounds violation (-6) from a clean return.

Algorithm 9.5 states the check abstractly, as the code generator thinks of it: a guard wrapped around the address computation for every index expression.

Algorithm 9.5: The bounds check the code generator inlines at every array index, together with the shared trap it jumps to on failure. The check is emitted per access; the trap is emitted once per program, gated by `needsBoundsCheck`.

Input: a base address $base$, a computed index i , an array length ℓ

Output: the element address $base + i$, or a trap

```

1  if  $i < 0$  then
2    | jump to L_BOUNDS_ERROR
3  if  $i \geq \ell$  then
4    | jump to L_BOUNDS_ERROR
5  return  $base + i$ 
6  at L_BOUNDS_ERROR: set  $R6 \leftarrow -6$ ; HALT

```

A safety choice C does not make. Standard C leaves an out-of-bounds array access *undefined*: the program may read a neighbouring variable, corrupt the stack, crash, or appear to work, and the compiler is entitled to assume it never happens [43]. That assumption enables optimisations and costs nothing at run time, and it is also the root of a large fraction of real security vulnerabilities. FRISCCc makes the opposite trade: it spends two comparisons and a branch on every access to guarantee that a bad index stops the program cleanly rather than silently corrupting it. For a language meant to be learned from, a deterministic halt is worth far more than the cycles it costs, and the optimiser can still prove some checks redundant and remove them — though FRISCCc, conservatively, keeps them all.

9.6 The runtime of `real_prime_sieve`

We can now watch the whole runtime assemble itself around the anchor program we have carried since [Chapter 7](#). The prime sieve fills a `char` array `isPrime` with ones, clears the multiples of each prime it finds, and counts the survivors. Two features of that source reach down into the runtime. The inner test `p * p <= 200` and the body line `j = p * p` multiply two integers, which sets `needsMul`; and the array accesses `isPrime[i]`, `isPrime[p]`, and `isPrime[j]` each generate a bounds check, which sets `needsBoundsCheck`. Nothing in the program divides, takes a modulus, or touches a float.

So the runtime FRISCCc appends to this program is exactly two things: the `F_MUL` routine of [Listing 4](#), and the `L_BOUNDS_ERROR` trap (a label plus two instructions) that the inline checks of [Listing 5](#) jump to. The other five integer-and-float helpers, and the whole fixed-point tower, are never emitted, because their flags were never set. This is the on-demand discipline of [Figure 9.1](#) in miniature: the program pays for the two operations it uses and not a byte more.

That economy has a run-time consequence we can hand directly to the next chapter. Because the sieve multiplies inside a loop, each trip through `F_MUL` costs its full shift-and-add loop, and the bounds checks fire on every one of the hundreds of array accesses the sieve performs. When [Chapter 10](#) loads this program into the FRISC simulator and counts cycles, a visible share of the total will be the runtime doing the work the source language took for granted — multiplying, and checking that every index was in range. The runtime is small to read and large to run, and that gap is exactly what the simulator was built to measure.

Practice

Exercise 9.1 (Applied · ★★☆☆)

Trace a multiply. Hand-execute `F_MUL` from Listing 4 on the inputs $a = 6$, $b = 5$. Write down the values of `R0` (the shifting copy of a), `R1` (the shrinking b), and `R3` (the accumulating result) at the top of each iteration of the loop labelled `L_MUL_LOOP_5` in Listing 4. How many times does the loop body run, and how many of those iterations execute the `ADD`?

Exercise 9.2 (Applied · ★☆☆)

Reading hexadecimal immediates. Three instructions in this chapter use hexadecimal immediates whose decimal value is easy to misread: `LOAD R1, (R5+0C)` in `F_MUL`, `MOVE 20, R4` in `F_DIV`, and `SHL R0, 10, R0` in `F_I2F`. Give the decimal value of each immediate and explain, in one sentence each, why that value is the right one for the routine's purpose.

Exercise 9.3 (Applied · ★★☆☆)

The cost of a guard. Count the instructions a single array access adds for its bounds check in Listing 5, not counting the index computation itself. `real_prime_sieve`'s first loop writes `isPrime[i] = 1` for i from 0 to 200. Ignoring the trap (which never fires on a correct run), how many extra instructions does bounds checking add across that one loop? What does this suggest about when a bounds-check elimination pass would pay for itself?

Exercise 9.4 (Applied · ★☆☆)

Why zero division returns zero. `F_DIV` returns 0 when asked to divide by zero, rather than trapping like a bounds violation does. Argue both sides: what would be gained by routing division-by-zero to a trap like `L_BOUNDS_ERROR`, and what would be lost? Your answer should mention the absence of any exception mechanism on FRISC.

Exercise 9.5 (Applied · ★★☆☆)

Fixed-point by hand. Using the Q16.16 format of Figure 9.4, give the stored 32-bit integer (in hexadecimal) for the values 1.0, 0.25, and -2.5 . Then verify that adding the stored integers for 0.25 and 0.25 yields the stored integer for 0.5 — that is, that Q16.16 addition is just integer addition.

Project

Exercise 9.6 (Lab · ★★★)

Project: a software multiply for tinyC. The companion repository’s `ch09-runtime` module is where you give tinyC the multiply that its target, like FRISC, lacks in hardware. The starter project in `frisc-cc-companion/ch09-runtime/starter/` contains a working code generator for the integer subset of tinyC, with one hole: the lowering of the `*` operator currently throws `UnsupportedOperationException`. Your job is to fill the hole the way FRISCcc does — by emitting a call to a software multiply helper, and by emitting the helper itself.

Concretely, implement two methods in `RuntimeEmitter.java`, each marked `// TODO: implement`. The first, `lowerMultiply`, should follow the call handshake of [Section 9.2](#): stage the two operands, open an argument area, copy them in, `CALL F_MUL`, clean the arguments, and move `R6` into the result register. The second, `emitMulHelper`, should emit the shift-and-add routine of [Algorithm 9.2](#), including the sign handling — tinyC’s `int` is signed, so a correct helper must handle negative operands. A `needsMul` flag (already wired into the emitter) should gate the helper so that programs without a multiply do not carry one.

When both methods are done, run

```
mvn test -pl ch09-runtime
```

from the companion root. The test suite multiplies a spread of operands — positive, negative, zero, and a few near the word boundary — through the full pipeline and simulator and checks the results against Java’s own multiplication, and it asserts that a multiply-free program emits no `F_MUL` label. The reference implementation in `ch09-runtime/solution/` is about ninety lines of Java; budget an evening, and keep [Listing 4](#) open beside you.

Stretch goal: add a compile-time fast path. When one operand is a constant power of two, emit a single `SHL` instead of the helper call, and confirm against the test suite that the result is unchanged. This is the smallest instance of the strength reduction that [Chapter 7](#) performs in general, and it is the first optimisation most real back ends apply to multiplication.

Your turn

Head to the runtime starter module in the companion repository and give your tinyC compiler a multiply. Implement `lowerMultiply` and `emitMulHelper` in `RuntimeEmitter.java`, following the call handshake of [Section 9.2](#) and the shift-and-add routine of [Algorithm 9.2](#). Run `mvn test -pl ch09-runtime` from the companion root; the multiplication spread and the no-multiply-no-helper assertion should all pass. When they do, your tinyC compiler can multiply — and you have written, in full, a piece of a runtime, the kind of code every compiler for every machine without a multiplier quietly carries.

Notes and further reading

The epigraph is Ken Thompson’s, and the runtime is brute force in his sense: where a bigger machine would have an instruction, FRISCCc has a loop that does the obvious thing one bit at a time. The obviousness is load-bearing — it is what lets us read and trust every routine — and it is also what the optimiser of [Chapter 7](#) spends its effort avoiding.

The algorithms in this chapter are the classical ones. Shift-and-add multiplication and restoring division are covered in Knuth’s *Seminumerical Algorithms* [51], the canonical reference for multiple-precision and machine arithmetic; the same volume treats the subtleties of signed division and the truncation conventions our `F_DIV` and `F_MOD` follow. For the bit-level tricks — the parity test by `AND`, the negate by subtract-from-zero, the double-width add by `ADC` that `F_FMUL` relies on — Warren’s *Hacker’s Delight* [82] is the modern standard reference, and it catalogues faster division algorithms (non-restoring, Newton–Raphson, and reciprocal-multiplication) than the simple restoring loop FRISCCc emits.

The decision to represent `float` as fixed-point rather than IEEE-754 is one that embedded and digital-signal-processing systems make routinely; Yates’s introduction [87] is a compact treatment of `Q`-format arithmetic and the scaling rules that [Algorithm 9.4](#) embodies. The full weight of what we chose *not* to implement is laid out in Goldberg’s survey [32], required reading for anyone who has ever been surprised by floating-point, and a good way to appreciate how much complexity `Q16.16` sidesteps.

The on-demand emission discipline of [Section 9.1.1](#) is FRISCCc’s version of a much larger idea: that a program should carry only the support code it uses. Production toolchains achieve this with linkers that pull individual object files out of an archive, and with the `libgcc` and `compiler-rt` runtime libraries that supply exactly the `__mulsi3`, `__divsi3`, and floating-point soft-float routines our helpers stand in for. The architectural context — why a RISC machine omits multiply and divide in the first place, and what that costs — is the subject of Patterson’s original RISC case [68] and the standard treatment in Hennessy and Patterson [35].

Finally, the semantic-preservation arguments that justify lowering an IR `mul` to a `CALL F_MUL`, and an array index to a guarded address computation, live in [Appendix G](#) alongside the optimisation proofs — the formal counterpart to this chapter’s claim that the runtime computes exactly what the source language meant.

Chapter 10

Running it: the simulator

“Beware of bugs in the above code; I have only proved it correct, not tried it.”
— Donald Knuth

For nine chapters we have been making promises. The lexer promised that a stream of characters meant a stream of tokens; the parser promised that those tokens meant a tree; the semantic analyser promised the tree was well typed; the lowering walker promised the typed tree meant a sequence of IR instructions; the optimiser promised the new IR meant the same thing as the old; and the code generator of [Chapter 8](#), together with the runtime of [Chapter 9](#), promised that the FRISC assembly in `a.out` meant exactly what `real_prime_sieve` meant when we wrote it in C. Every one of those promises was a theorem about meaning, and meaning, in a compiler, is the thing you cannot see. A register-allocated, frame-disciplined, helper-calling FRISC program is a long argument that something is true. The simulator is where we finally make the machine settle it.

This is the last chapter of the back end, and its job is the most concrete one in the book: take the bytes we generated and run them. Not on a FRISC chip — there is no FRISC chip on your desk, and as we will see in a moment that is a feature rather than a limitation — but on *FRISCjs*, a pure-JavaScript simulator that implements the FRISC instruction set one instruction at a time. The program we run is `real_prime_sieve`: it marks composites in a 201-entry boolean array with the sieve of Eratosthenes and returns the count of primes at most 200. We know what the answer should be. There are exactly 46 primes not exceeding 200. By the end of this chapter the simulator will have told us, in 44,355 machine cycles, that the answer is 46, and we will know precisely how it got there: which instructions it fetched, in what order, how many times, and where the cycles went.

A simulator is a humble program with a grand role. It is the referee, the oracle, and the microscope all at once. As referee it decides whether the compiler kept its promises. As oracle it is the only authority we have on what a FRISC program *does*, because the instruction set is defined by what the simulator does with it. And as microscope it lets us watch a running program at a resolution no real processor would ever expose: every register, every memory write, every branch, frozen between cycles for inspection. The rest of this chapter is about how that humble program works, how our compiler drives it, and how to read what it tells us.

10.1 Why a simulator, not silicon

FRISC is a teaching architecture. It was designed at the Faculty of Electrical Engineering and Computing in Zagreb to be simple enough that a student can hold the whole instruction set in their head and implement a working processor for it in a lab, and that pedagogical origin shows in every corner of the design: eight general-purpose registers, a flat byte-addressed memory, a single condition-code register, fixed-width 32-bit instructions, and no multiply or divide in hardware. There is no commercial FRISC silicon to buy, and that is exactly the point. The architecture exists so that the path from source code to running instructions can be understood completely, and a simulator preserves that completeness in a way that real hardware never could.

Consider what running on real silicon would cost us. A physical processor is a black box with pins. We can put a program in and read a result out, but we cannot stop it between two instructions and ask what every register holds, because stopping it perturbs it. We cannot replay the same run twice and get bit-for-bit identical behaviour once caches, interrupts, and memory timing enter the picture. And we certainly cannot count, exactly, how many times the inner sieve loop executed a `LOAD`. A simulator gives us all three for free. It is *observable*: the entire machine state is a JavaScript object we can read at any moment. It is *deterministic*: the same program loaded into the same initial memory produces the same cycle-by-cycle trace every single time. And it is *instrumentable*: we can hang a counter, a logger, or a breakpoint off any event in the execution loop without changing the program's result.

Key insight

A simulator is an interpreter whose source language is machine code. It reads an instruction, works out what the instruction means, and performs that meaning on a model of the machine's state — the identical *read, decode, act* loop we will build for the typed IR in [Chapter 11](#) and for a bytecode in [Chapter 12](#). The only thing that makes a simulator feel different from those interpreters is the language it interprets: fixed-width binary words instead of tree nodes or bytecodes. Strip away the encoding and a CPU *is* an interpreter, etched in silicon instead of written in JavaScript.

That last observation is worth dwelling on, because it is the bridge from this part of the book to the next. We have spent the back end *compiling*: translating ahead of time into a lower language so that execution is cheap. The simulator reminds us that at the very bottom, something always interprets. The FRISC chip a student wires up in a lab interprets 32-bit words; FRISCjs interprets the same words in software; and in Part IV we will deliberately move the interpretation boundary back up, executing the IR directly rather than lowering it. Compilation and interpretation are not rival philosophies. They are a choice about where to put the interpreter, and the simulator is the interpreter we have been compiling *toward* all along.

10.2 The fetch–decode–execute cycle

A processor, real or simulated, runs one loop forever, or until something tells it to stop. It reads the instruction that the program counter points at, works out what that instruction is, does what the instruction says, advances the program counter, and goes back to the top. This is the *fetch–decode–execute cycle*, the oldest idea in computing and the heartbeat of the von Neumann machine. Everything a FRISC program does — every arithmetic result, every loop, every function call — is some number of trips around this loop.

In FRISCjs the loop body is a single method, `performCycle`, and the whole of execution is nothing more than calling it repeatedly. [Algorithm 10.1](#) states it in the simulator’s own terms. The program counter `PC` names a memory address; the simulator reads a 32-bit word from there, decodes it into an operation and a list of arguments, dispatches to the handler for that operation, and then — crucially — advances `PC` by four, because FRISC instructions are four bytes wide and the default flow of control is to fall through to the next one.

Algorithm 10.1: One trip around the FRISC execution loop, as implemented by `performCycle` in FRISCjs. Execution is this procedure called repeatedly until a `HALT` instruction stops the clock.

Input: A simulator state: register file R (including `PC`), condition register `SR`, and memory `MEM`.

Output: The state advanced by exactly one instruction, or a halt.

```

1  word ← MEM.read( $R.PC$ )                (fetch four bytes at  $PC$ )
2
3  ( $op$ ,  $args$ ) ← decode( $word$ )            (see Algorithm 10.2)
4
5  if  $op$  is undefined then
6    stop()
7    raise “illegal instruction”
8  handler[ $op$ ]( $args$ )                    (execute: mutate  $R$ ,  $SR$ , or  $MEM$ )
9
10  $R.PC$  ←  $R.PC + 4$                       (advance to the next word)
11
12 acceptInterrupt()                      (poll I/O lines; unused by our programs)
13
```

Two details in [Algorithm 10.1](#) repay attention because they explain behaviour we will see in the trace later. The first is the unconditional `PC` increment. The handler runs *before* `PC` advances, which means a jump instruction cannot simply set `PC` to its target: if it did, the increment that follows would overshoot the target by four. FRISCjs resolves this the way real FRISC does, by having jumps and calls write *target minus four* into `PC`, so that the unconditional “+4” lands exactly on the target. It is a small accounting trick, and forgetting it is the classic off-by-one bug in any instruction-set simulator. The second detail is that

the cycle ends by polling the interrupt lines. FRISC supports maskable and non-maskable interrupts driven by memory-mapped peripherals, and a faithful simulator must check them every cycle. Our compiler emits no interrupt-driven code — the programs in this book are pure computations that start at `main` and end at `HALT` — so for us the poll always finds nothing and falls through. We mention it because it is in the loop, and a chapter that claimed the loop had only three steps would be lying about the machine.

The cycle is also where the word “cycle” earns its double meaning, and we should be honest about it from the start. On real FRISC silicon, different instructions take different numbers of clock ticks; a `LOAD` that goes to memory is slower than an `ADD` between two registers. FRISCjs makes no attempt to model that. One call to `performCycle` retires exactly one instruction, whatever the instruction is, so when we “count cycles” in this chapter we are really counting *retired instructions*. That is a coarse model of time, but it is the right model for a compiler writer: the number of instructions a program executes is something the compiler controls directly, whereas per-instruction timing is a property of a hardware implementation we do not have. We will return to this distinction in [Section 10.8](#), and lean on it in [Chapter 14](#).

10.3 Inside FRISCjs

FRISCjs is two cooperating pieces: an *assembler* that turns FRISC source text into a memory image, and a *simulator* that executes a memory image. Our compiler produces assembly text in `a.out`, so both pieces are in play, and it is worth seeing how they fit together before we drive them. [Figure 10.1](#) lays out the whole arrangement, including the Java harness our compiler wraps around it, which we build in [Section 10.6](#).

The assembler’s job is the inverse of the code generator’s. Where [Chapter 8](#) turned IR into mnemonic text like `ADD R1, R2, R3`, the assembler turns that text back into the 32-bit binary words a processor actually fetches, resolving labels to addresses along the way. Its output is a *memory image*: an array of eight-character binary strings, one per byte, ready to be copied into the simulator’s memory starting at address zero. We never look at this image directly — it is an implementation detail between the assembler and the simulator — but its existence is why the simulator can begin executing at `PC = 0` the moment the image is loaded.

The simulator proper is an object with two components. The first, `MEM`, models memory as a flat array of bytes with helpers to read and write one, two, or four bytes at a time. FRISC is little-endian, so a 32-bit word stored at an address keeps its least significant byte at the lowest address; the simulator’s read assembles a word by shifting byte 0 into bits 0–7, byte 1 into bits 8–15, and so on. Word and half-word accesses are forced to be aligned — the simulator masks the low bits of the address with `& ~0x03` for a word and `& ~0x01` for a half-word — which matches the code generator’s assumption that every slot in a stack frame is word-aligned. The second component, `CPU`, holds the processor state: the eight general-purpose registers `R0` through `R7`, the program counter `PC`, the status register `SR` that

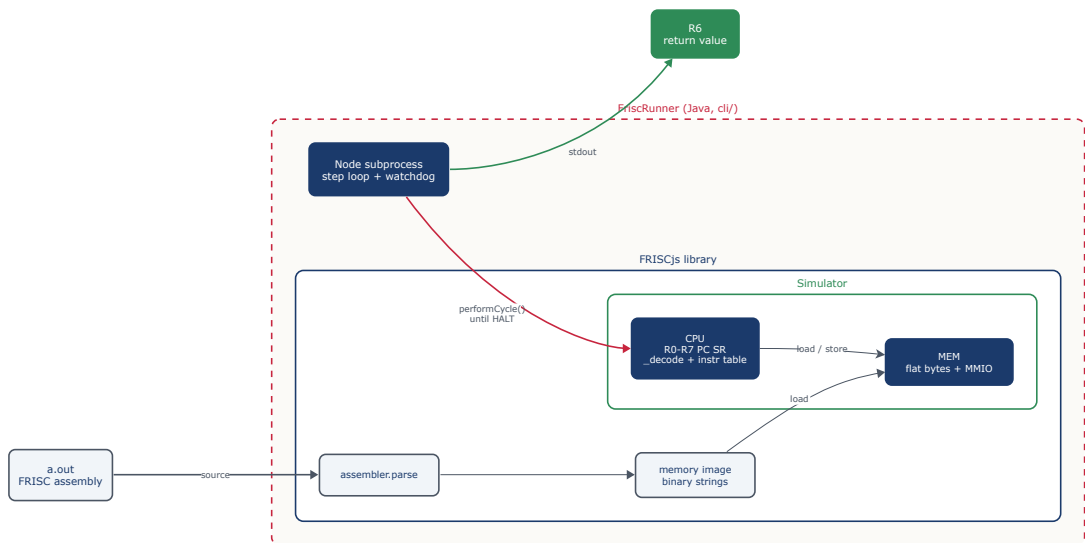


Figure 10.1: The simulator in context. Our Java `FRiscRunner` spawns a Node subprocess that loads the FRISCjs library, hands `a.out` to the assembler to produce a memory image of binary strings, loads that image into a fresh `Simulator`, and steps the CPU component until it halts. The CPU and the memory (MEM) are the simulator's two halves; the CPU fetches and stores through MEM, and the return value of `main` is read out of register R6 on standard output.

carries the condition flags, and an internal interrupt-enable bit. It also holds the decode logic and the table of instruction handlers — the machinery of [Algorithm 10.1](#).

Whose simulator is this? FRISCjs is an open-source project written by Ivan Zuzak and others, originally to give the Zagreb PLT and digital-logic courses a browser-based FRISC environment with a register display, a memory view, and a single-step button. It ships as a JavaScript library with a clean separation between the assembler and the simulator, and that separation is what lets us embed it headlessly: we ignore the graphical front end entirely and call the library functions directly from a script. Everything this chapter says about FRISC’s behaviour is, ultimately, a statement about what this library does — which is the appropriate kind of authority for a teaching architecture, where the reference implementation *is* the specification.

The status register deserves a closer look, because it is where the results of comparisons live and therefore where conditional control flow is decided. SR packs the four condition flags into its low bits: Z (zero, the result was zero), N (negative, the result’s sign bit was set), C (carry, an arithmetic operation carried or borrowed out of the top bit), and V (overflow, a signed operation overflowed). The arithmetic and compare instructions set these flags as a side effect; the conditional jumps read them. When the code generator emitted `CMP R0, R1` followed by `JP_SLE` to implement `p * p <= 200`, it was relying on `CMP` to set N, V, and Z from the subtraction `R0 - R1`, and on the simulator’s condition test for “signed less-or-equal” to read exactly the combination (`N ≠ V`) or Z that the FRISC manual prescribes. The flags are the thin interface between computing a comparison and acting on it, and the simulator implements both ends.

10.4 Decoding an instruction

Fetching a word is easy; the word is just an integer. Decoding it — working out that a particular pattern of bits means “add R1 and R2, put the result in R3” — is where the instruction set’s encoding becomes real. A FRISC instruction is a 32-bit word carved into fields, and decoding is the act of slicing the word back into its fields and reading each one. [Figure 10.2](#) shows the carve-up for the arithmetic-and-logic format, using a word our own compiler could have emitted.

The encoding is regular by design. The opcode always lives in the top five bits, bits 27 through 31, so the very first thing the decoder does is read those five bits and look them up in a table that maps `00100` to `ADD`, `10110` to `LOAD`, `11001` to `CALL`, and so on for the twenty-seven instructions FRISC defines. Once the opcode is known, its format is known, and the format dictates which other bit-ranges carry meaning. For the arithmetic and logic instructions the destination register is bits 23–25, the first source is bits 20–22, and bit 26 is a mode flag: when it is zero the second source is the register named in bits 17–19, and when it is one the second source is a 20-bit signed immediate sitting in bits 0–19, sign-extended to a full word. [Algorithm 10.2](#) gives the decoder in full for the formats our compiler actually emits.

Algorithm 10.2: Decoding a FRISC word, following the structure of FRISCjs's `_decode`. Here `bits( $w, a, b$ )` extracts bits a through b of w as an unsigned integer, and `signExtend( $\cdot$ )` treats a 20-bit field as a signed two's-complement number.

Input: A 32-bit instruction word w .

Output: An operation name op and its argument list, or $(\perp, \perp)$ if the opcode is unknown.

```

1   $op \leftarrow opcodeTable[bits(w, 27, 31)]$ 
2  if  $op$  is undefined then
3    return  $(\perp, \perp)$ 

4  if  $op$  is arithmetic/logic (ADD, SUB, AND, SHL, ...) then
5     $dest \leftarrow reg[bits(w, 23, 25)]$ 
6     $src1 \leftarrow reg[bits(w, 20, 22)]$ 
7    if  $bits(w, 26, 26) = 0$  then
8       $src2 \leftarrow reg[bits(w, 17, 19)]$ 
9    else
10      $src2 \leftarrow signExtend(bits(w, 0, 19))$            (20-bit immediate)
11
12     $args \leftarrow (src1, src2, dest)$ 
13 else if  $op$  is memory (LOAD, STORE, LOADB, ...) then
14    $r \leftarrow reg[bits(w, 23, 25)]$ 
15    $base \leftarrow bits(w, 26, 26) = 0 ? 0 : reg[bits(w, 20, 22)]$ 
16    $off \leftarrow signExtend(bits(w, 0, 19))$ 
17    $args \leftarrow (r, base, off)$ 
18 else if  $op$  is control (JP, CALL, JR, RET, HALT) then
19    $cond \leftarrow condTable[bits(w, 22, 25)]$            (predicate on the flags)
20
21   extract target or return mode as the format requires
22    $args \leftarrow (cond, \dots)$ 
23 else
24   decode MOVE, POP, PUSH, CMP analogously
25 return  $(op, args)$ 

```

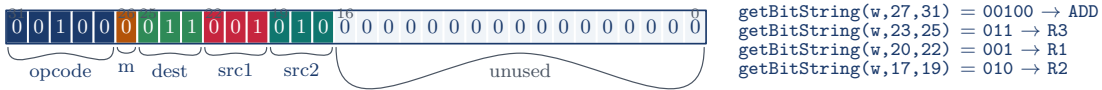


Figure 10.2: The 32-bit word that FRISC’s assembler emits for `ADD R1, R2, R3`, sliced into its fields. The top five bits are the opcode; bit 26 chooses whether the second source is a register or an immediate; three-bit fields name the destination and the two source registers. The decode column on the right shows the exact bit-range each field occupies, which is what the simulator’s bit-extraction routine reads. The seventeen low bits are unused in the register–register form.

The shape of [Algorithm 10.2](#) is a cascade of opcode classes, and that shape is not arbitrary: it mirrors the instruction formats exactly. Every arithmetic and logic instruction is decoded the same way because they share a format; every memory instruction shares a different format; the control-transfer instructions share a third in which bits 22–25 carry a condition code rather than register numbers. The decoder is, in effect, the encoding table read backwards. This is why a teaching ISA keeps its formats few and regular — not to make the hardware cheap, though it does that too, but to make the decoder small enough to understand. A real production ISA with dozens of formats and variable-length encodings needs a decoder that is itself a small compiler; FRISC’s decoder is a single readable function.

One subtlety hides in `signExtend`. The immediate and offset fields are twenty bits, but the values they represent — a loop bound like 200, a frame offset like -20 — are full 32-bit quantities. Sign extension copies the field’s top bit into all the bits above it, so a field whose top bit is set (a negative offset, as every stack-relative address in our frames is) becomes a negative 32-bit integer, and a field whose top bit is clear stays positive. Without it, `SUB R7, 0x10C, R7` — the prologue instruction that allocates 268 bytes of locals for `main` — would still work, because `0x10C` is positive, but `LOAD R0, (R5-20)` would read from the wrong address, because the offset -32 would be misread as a large positive number. The code generator and the simulator agree on the encoding precisely because both implement the same twenty-bit signed convention.

10.5 Memory, addresses, and how a program stops

We have a loop and a decoder; we need somewhere for the program to live and a way for it to end. Both are properties of the memory model, so this is the place to make the model precise.

FRISC memory is flat and byte-addressed. There is no virtual memory, no protection, no segments — one contiguous array of bytes from address zero upward, and the program shares it with its own stack and its global data. Our compiler partitions this space by convention, not by enforcement. Global arrays like `isPrime` are laid out at fixed addresses; the program’s code occupies the bytes the assembler placed it in starting at zero; and the

stack grows *downward* from a high address, initialised by the very first instruction the compiler emits, `MOVE 40000, R7`, which sets the stack pointer to byte `0x40000` (262,144 in decimal — FRISC assembly literals are hexadecimal by default). Every function prologue pushes the old frame pointer and subtracts from `R7` to carve out room for locals, exactly as [Chapter 9](#) described; the simulator simply watches `R7` march down toward the data and trusts the compiler not to let the stack and the globals collide. On a teaching machine with no memory protection, that trust is the whole safety story, which is one more reason the bounds-checking helper of [Chapter 9](#) earns its keep.

Reading and writing go through a single chokepoint, and that chokepoint is more interesting than it first appears. Before every access the memory component asks whether the address falls inside a region claimed by a *memory-mapped I/O* unit. FRISC peripherals — a timer, a serial port, a counter — are not reached through special instructions; they are wired into the address space, so that storing to a particular address is how you talk to the device and loading from it is how you read the device’s state. If the address overlaps such a region, the access is routed to the device rather than to plain memory; otherwise it lands in the byte array.

The I/O we do not use. FRISCjs models memory-mapped peripherals faithfully — a programmable timer, interrupt controllers, generic I/O units — and the digital-logic course uses them heavily, blinking lights and counting pulses. Our compiler uses none of it. A C-subset program that computes a value and returns it never touches a device address, so for our programs the I/O overlap check always fails and every access is a plain memory access. We flag this deliberately: it would be easy to read the simulator and conclude that I/O is central, when in fact it is machinery our particular workload leaves idle. The peripherals are there; we just do not knock on their doors.

That leaves the question of how a program ends, and the answer ties the whole back end together. A FRISC program stops when the CPU executes a `HALT` instruction, which clears the run flag and, in our harness, sets a “halted” marker the driver loop checks. Our compiler emits exactly one `HALT`, in the three-instruction bootstrap it places at address zero:

```

1      MOVE 40000, R7      ; initialise stack pointer (SP)
2      CALL F_MAIN        ; call main
3      HALT               ; program end

```

The `CALL` transfers control to `main`; when `main` executes its `RET`, control returns to the instruction after the `CALL`, which is the `HALT`. So the program halts precisely when `main` returns, and not before. And by the ABI of [Chapter 9](#), `main`’s return value is sitting in register `R6` at that moment. The simulator does not know or care that `R6` is special — to the CPU it is just one of eight registers — but our convention makes it the program’s answer, and the harness reads it out the instant the machine halts. The bridge from “the machine stopped” to “the program returned 46” is one register and one convention.

10.6 Driving the simulator from the compiler

FRISCjs is a library, not a command. To run a program through it we have to write a small amount of glue: load the library, hand it the assembly, build a simulator, step it, and read out the result. In FRISCcc that glue lives in a Java class, `FrisccRunner`, which spawns a Node subprocess and feeds it a short JavaScript driver. The heart of that driver is reproduced below.

```

1  const source    = fs.readFileSync(frisccPath, 'utf8');
2  const parsed    = asm.parse(source);           // assemble to a memory image
3  const simulator = new Simulator();
4  simulator.MEM._size = memSizeKb * 1024;
5  simulator.MEM.loadBinaryString(parsed.mem);    // load image at address 0
6
7  let halted = false;
8  simulator.CPU.onStop = function () { halted = true; };
9
10 let steps = 0;
11 while (!halted && steps < stepLimit) {
12     simulator.CPU.performCycle();               // one instruction
13     steps += 1;
14 }
15
16 if (!halted) {
17     console.error('Execution step limit exceeded: ' + stepLimit);
18     process.exit(124);
19 }
20 console.log(String(simulator.CPU._r.r6));      // report R6

```

Listing 6: The core of FRISCcc’s `FrisccRunner` driver: assemble, load, step to HALT under a watchdog, and print register R6. The full class adds a process timeout and parses the printed value back into a Java result object.

The structure of this loop is worth stating as an algorithm in its own right, because the watchdog at line 10 is doing real work. [Algorithm 10.3](#) abstracts it.

The step limit is not a performance knob; it is a halting-problem concession. We cannot decide in general whether an arbitrary FRISC program will ever execute HALT, and a compiler bug or an infinite loop in a test program would otherwise spin `performCycle` forever and freeze whatever is running the suite. So the driver caps the run at a generous two hundred million instructions — comfortably more than any program in our examples needs, and far short of an afternoon — and treats exhaustion as a distinct, reportable failure rather than a crash. When a test that should return 46 instead trips the watchdog, that is itself diagnostic information: it says “your program does not terminate,” which is a different bug from “your program returns the wrong number.”

Algorithm 10.3: How FRISCcc runs a compiled program: step the simulator until it halts, but give up after L instructions so that a non-terminating program fails loudly instead of hanging the build.

Input: A path to a FRISC assembly file, a memory size, and a step limit L .

Output: The value of R6 at halt, or a failure if the program does not halt within L steps.

```

1  image  $\leftarrow$  assemble(read(path))
2  sim  $\leftarrow$  new Simulator()
3  sim.load(image)                                (copy image into MEM at address 0)
4
5  halted  $\leftarrow$  false;  steps  $\leftarrow$  0
6  set sim.onStop to assign halted  $\leftarrow$  true          (HALT fires this)
7
8  while  $\neg$ halted and steps  $<$   $L$  do
9    | sim.performCycle()
10   | steps  $\leftarrow$  steps + 1
11 if  $\neg$ halted then
12   | return FAILURE("step limit  $L$  exceeded")          (the program looped forever)
13   |
14 else
15   | return sim.R[R6]
```

Why we step the loop ourselves. FRISCjs ships a convenient `run()` method that executes the program for us. We do not use it, and the reason is a small performance scandal worth recording. `run()` schedules `performCycle` on a JavaScript timer, one cycle per timer tick, because the library was built for a browser where you want the register display to animate as the program runs. A timer tick has a floor of a millisecond or so, so `run()` executes at best a few hundred instructions per *second* — fine for watching a ten-instruction demo single-step in a classroom, catastrophic for a 44,355-cycle sieve, let alone a benchmark suite. Our tight `while` loop calls `performCycle` back to back with no timer in between, and runs the same sieve in well under a second. The lesson generalises: a simulator’s *stepping mechanism* and its *execution engine* are separate concerns, and an interface designed for interactive single-stepping is the wrong interface for batch execution. When embedding someone else’s simulator, it is worth checking how its main loop is scheduled before trusting its throughput.

10.7 Reading a cycle trace

Because the simulator exposes its whole state between cycles, we can watch a program run. FRISCjs fires an `onBeforeExecute` callback just after it decodes each instruction and just before it executes it, and an `onAfterCycle` callback once the instruction has run. Hanging a logger off these gives us a *cycle trace*: a record of which instruction ran at each step and what it changed. Table 10.1 is the real trace of the first twelve cycles of `real_prime_sieve`, captured exactly this way — the bootstrap and the prologue of `main`, the moment a program comes to life.

Read top to bottom, Table 10.1 tells the story of the ABI in motion. Cycle 0 plants the stack pointer at `0x40000`. Cycle 1 is the `CALL`: notice that `R7` drops by four — the call pushed the return address onto the stack — and that the next instruction fetched is at `0x0C`, not `0x08`, because the call jumped over the `HALT` at `0x08` straight into `F_MAIN`. (The `HALT` is not dead; it is where the eventual `RET` from `main` will land.) Cycles 2 through 4 are the textbook prologue: save the caller’s frame pointer, establish our own, and subtract `0x10C` — 268 bytes — from the stack pointer to reserve room for `main`’s locals and temporaries. From cycle 5 the program is doing its own work, setting up a five-iteration loop to zero the local slots before the sieve begins. Every later cycle of the program’s 44,355 is more of the same: fetch, decode, change one register or one word, advance. The machine has no plans and no memory of its past; it only ever does the next instruction. That a sieve of Eratosthenes emerges from forty thousand such steps is the entire magic of a stored-program computer, and the trace is where you can watch the magic refuse to be magic.

A trace like this is also the back end’s debugger. When a generated program returns the wrong answer, the cycle trace is where we find out why: we run it, we watch the registers, and we find the exact instruction at which the machine’s state diverges from what we expected. More than one bug in FRISCcc’s code generator was caught by staring at a trace

#	PC	Instruction	Effect	Meaning
0	0x00	MOVE 40000, R7	$R7 \leftarrow 0x40000$	set SP to byte 0x40000
1	0x04	CALL F_MAIN	$R7 \leftarrow 0x3FFFC$	push return addr, jump
2	0x0C	PUSH R5	$R7 \leftarrow 0x3FFF8$	save caller's FP
3	0x10	MOVE R7, R5	$R5 \leftarrow 0x3FFF8$	set this frame's FP
4	0x14	SUB R7, 10C, R7	$R7 \leftarrow 0x3FEEC$	allocate 268 bytes of locals
5	0x18	MOVE 5, R1	$R1 \leftarrow 0x5$	loop counter for zeroing
6	0x1C	MOVE R5, R0	$R0 \leftarrow 0x3FFF8$	base for zeroing locals
7	0x20	SUB R0, 14, R0	$R0 \leftarrow 0x3FFE4$	first slot to clear
8	0x24	MOVE 0, R2	—	the zero value
9	0x28	STORE R2, (R0)	$MEM[0x3FFE4] \leftarrow 0$	clear one word
10	0x2C	ADD R0, 4, R0	$R0 \leftarrow 0x3FFE8$	advance to next word
11	0x30	SUB R1, 1, R1	$R1 \leftarrow 0x4$	one fewer word to clear

Table 10.1: A real cycle trace of the program coming to life: the three-instruction bootstrap (cycles 0–1), the prologue of main (cycles 2–4), and the start of the loop that zero-initialises the local slots (cycles 5 onward). The Effect column shows only the register or memory location each instruction changed; cycle 8 changes nothing because R2 already held zero.

and noticing a frame offset that was four bytes wrong, or a comparison that set the flags from the wrong operand order. The simulator's observability is not a luxury; it is how the back end is made trustworthy in the first place.

10.8 Counting cycles, counting instructions

The same callback that produced the trace can produce statistics. If, instead of logging each instruction, we tally it by opcode, the run yields a *dynamic instruction mix*: how many times each kind of instruction actually executed, as opposed to how many of each kind appear in the program text. The distinction matters enormously. The inner sieve loop is a handful of instructions in `a.out`, but it runs thousands of times, so its instructions dominate the mix far out of proportion to their footprint. Static code size tells you what the compiler emitted; the dynamic mix tells you what the machine did. Figure 10.3 is the dynamic mix for `real_prime_sieve`, every count exact.

The shape of Figure 10.3 is the most important empirical fact in the back end, and it is worth saying plainly: `real_prime_sieve` spends most of its life moving data, not computing on it. LOAD alone is 24% of the run; LOAD and STORE together are 42%; throw in the PUSH and POP of the calling convention and memory traffic crosses half of all cycles. Actual arithmetic — ADD, SUB, CMP, the shifts — is the remainder, and the genuinely arithmetic-heavy operations the program is “about” are a sliver. The mix is a measurement of an old truth: the program manipulates a large amount of data in a very small number of ways, and the ways that dominate are the duller ones, the loads and stores that shuttle values between registers and slots.

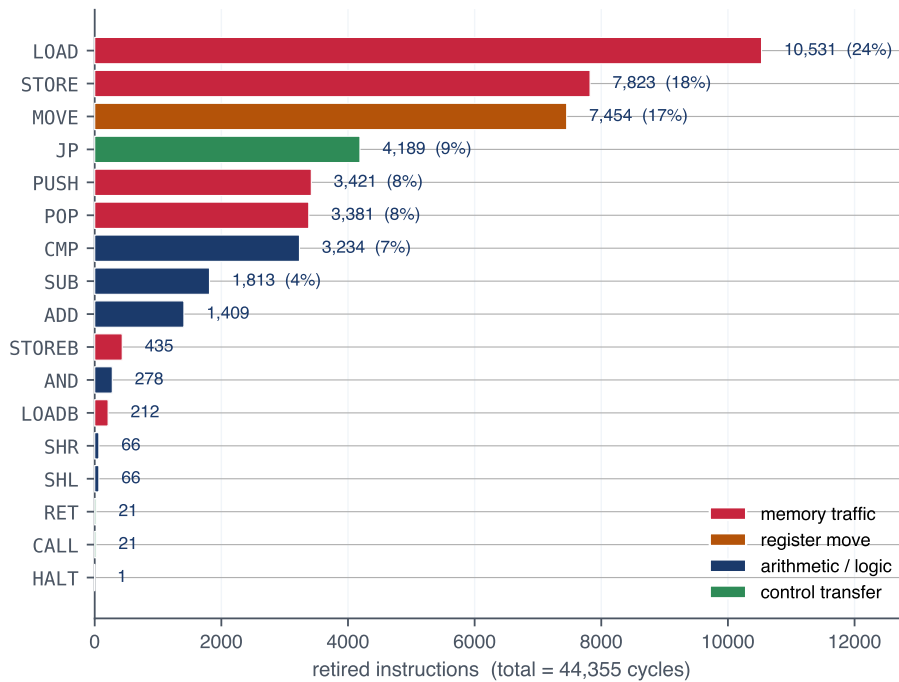


Figure 10.3: Every instruction the sieve retired in its 44,355-cycle run, tallied by opcode and coloured by category. Memory traffic — LOAD, STORE, PUSH, POP and their byte variants — accounts for well over half of all cycles; the single most frequent instruction, LOAD, is nearly a quarter of the run on its own. Arithmetic, the work the program ostensibly exists to do, is a minority.

This is exactly why the optimiser of [Chapter 7](#) aimed so many of its passes at memory — load forwarding, dead-store elimination, copy propagation — and it is the hook for the performance story of [Chapter 14](#). To see the optimiser’s effect at the level that finally matters, the machine, we can run the same program compiled two ways and compare cycle counts directly. [Table 10.2](#) does this for `real_prime_sieve`.

	-O0	-O1	Δ
Total cycles to halt	45,671	44,355	−1,316
JP (branches)	4,902	4,189	−713
STORE	8,024	7,823	−201
MOVE	7,655	7,454	−201
SUB	2,014	1,813	−201
LOAD	10,531	10,531	0
Result in R6	46	46	—

Table 10.2: The optimiser’s effect measured where it lands: retired instructions on the simulator. The `−O1` pipeline saves 1,316 cycles, just under three percent, and — the part that matters more than the percentage — returns the identical answer. Most of the saving is branches the control-flow simplifier removed; the rest is a handful of redundant stores, moves, and address computations. The unchanged LOAD count is honest: the sieve’s loads are genuine reads of the boolean array, and no local optimisation can conjure them away.

Two things about [Table 10.2](#) deserve comment, one modest and one important. The modest one is the size of the win: just under three percent. We should not oversell it. FRISCcc’s lowering walker already emits fairly compact IR, the example is small, and a three-percent cycle reduction on a teaching compiler is an honest number, not a disappointing one — the chapter on performance will put it in context against the larger wins available when the program structure gives the optimiser more to chew on. The important thing is the bottom row: R6 is 46 under both settings. Every pass in the optimiser carried a semantic-preservation obligation, and [Table 10.2](#) is that obligation discharged at the only court that has final say. The optimised program is faster and means the same thing, and the simulator is what lets us assert both at once with a straight face.

It bears repeating that the cycles in these tables are retired instructions, not silicon clock ticks. A real FRISC implementation would weight a memory LOAD more heavily than a register ADD, and on such a machine the memory-traffic story of [Figure 10.3](#) would look even starker, because the cheap instructions are exactly the ones the program runs least. Our instruction-count model understates the cost of memory rather than overstating it, which is the safe direction to be wrong in for a compiler whose passes are aimed at memory in the first place. When [Chapter 14](#) compares the compiler against the IR interpreter and the bytecode VM, it does so in this same currency — instructions retired — precisely so the three execution strategies can be set side by side on one axis.

10.9 Worked example: `real_prime_sieve` under the simulator

We have met every piece; let us run the whole thing once, end to end, the way we would at a terminal. Starting from the C source, one command compiles the program to FRISC assembly and hands it to the simulator:

```

1  $ ./run.sh --frisc --01 --run examples/real_world/real_prime_sieve/program.c
2  ... compiling ...
3  46

```

The lone 46 on standard output is register R6 at the moment of HALT, read out by the driver of [Algorithm 10.3](#) — the number of primes at most 200, which is the answer the program was written to compute. Behind that single line is everything this chapter described. The compiler turned the source into the 513-line `a.out` we have been studying; the FRISCjs assembler turned that text into a memory image of binary words; the driver loaded the image at address zero and called `performCycle` in a tight loop; and 44,355 cycles later the bootstrap’s HALT stopped the clock with 46 in R6.

If we want to see the work rather than just the answer, the same program run under an instrumented driver — the one that fed [Table 10.1](#) and [Figure 10.3](#) — reports the totals directly:

```

1  HALTED: true
2  R6:      46
3  CYCLES: 44355
4  top instructions:
5      LOAD   10531   STORE   7823   MOVE    7454
6      JP     4189   PUSH    3421   POP     3381
7      CMP    3234   SUB     1813   ADD     1409

```

Every number on that screen is a fact about a specific run of a specific program, reproducible to the cycle. That reproducibility is the quiet triumph of the whole exercise. We began the book with a string of characters and a claim about what it meant; we end the back end with a count of instructions and a register holding 46, and a chain of phases connecting the two that we can walk in either direction. The simulator did not make the compiler correct — the phases did that, each keeping its promise to the next. The simulator is how we *know* they did.

Practice

Exercise 10.1 (Applied · ★☆☆)

Follow the off-by-four. A CALL to a label at address 0x1C0 must leave PC equal to 0x1C0 after the cycle completes. Given that the handler runs before the unconditional $PC \leftarrow PC + 4$ of [Algorithm 10.1](#), what value must the CALL handler write into PC? Confirm your answer against the bootstrap trace in [Table 10.1](#), where the call at 0x04 lands at 0x0C.

Exercise 10.2 (Applied · ★☆☆)

Decode by hand. A FRISC word has opcode field 01101 (which is CMP), source-one field 000, mode bit 0, and source-two register field 001. Using [Algorithm 10.2](#), name the operation and its operands. Which condition flags will it set, and from what subtraction?

Exercise 10.3 (Applied · ★☆☆)

Predict the mix. Without running anything, sketch how the dynamic instruction mix of math_fibonacci_iter — an int-only loop with no array — should differ from [Figure 10.3](#). Which category should shrink most, and why?

Exercise 10.4 (Applied · ★☆☆)

Why does LOAD not move? In [Table 10.2](#) the LOAD count is identical at $-O0$ and $-O1$, while STORE, MOVE, and JP all fall. Give a one-sentence reason rooted in what the sieve's loads actually read, and name one program structure for which a load *could* be eliminated by the optimiser.

Exercise 10.5 (Applied · ★☆☆)

Where do the cycles go? The watchdog in [Algorithm 10.3](#) caps a run at two hundred million instructions. The sieve uses 44,355. Estimate, to an order of magnitude, the largest input array a sieve like this could handle before approaching the cap, and state which instruction category would grow fastest as the array grows.

Project

Exercise 10.6 (Lab · ★★☆☆)

Project: a cycle-counting harness. Your tinyC compiler already emits FRISC assembly; in `frisc-companion/ch09-runtime/` you have a runner that executes it and checks the returned value. In this project you turn that runner into an instrument. Extend the companion’s simulator driver so that, in addition to reporting the result in R6, it reports the total cycle count and a dynamic instruction mix — the same data behind [Figure 10.3](#). The starter in `ch09-runtime/starter/` gives you a driver that loads a program and steps it to HALT; your tasks are four.

1. Register an `onBeforeExecute` (or equivalent) callback that tallies one count per retired instruction, keyed by opcode, plus a single counter for total cycles.
2. On halt, print the cycle total and the per-opcode counts sorted descending, so the output matches the worked example’s instrumented form.
3. Add the watchdog: if the program does not halt within a configurable step limit, fail with a distinct “did not terminate” status rather than hanging or returning a wrong value.
4. Run your harness on a tinyC program compiled at your optimiser’s `-O0` and `-O1` levels and produce a small table in the shape of [Table 10.2](#). Confirm the result in R6 is identical at both levels — if it is not, you have found an optimiser bug, and the cycle trace is where to start looking.

The tests in `ch09-runtime/tests/` check that your harness reports the correct result and a plausible, monotonic cycle count — an unoptimised program must not retire *fewer* instructions than its optimised form. When the table balances and the answers match, you have built, in miniature, the measurement apparatus this entire chapter runs on.

Notes and further reading

The fetch–decode–execute cycle is the von Neumann architecture’s defining loop, set out in von Neumann’s 1945 report on the EDVAC [\[65\]](#); every simulator in this chapter is a direct descendant of that one idea. The framing of a CPU as an interpreter of machine code, and the consequent kinship between simulators and the tree-walkers and bytecode machines of Part IV, is the organising theme of Nystrom’s *Crafting Interpreters* [\[67\]](#); we will lean on it heavily in [Chapter 11](#) and [Chapter 12](#).

FRISCjs itself is the reference we have been treating as the FRISC specification; its assembler and simulator are the authority for the behaviour described here, and the complete FRISC instruction encoding, addressing modes, and flag semantics are catalogued in [Appendix B](#), with the simulator’s invocation and output format in [Appendix C](#). Readers who want to see the same ideas at a larger scale will find the SPIM and MARS simulators for MIPS [\[52\]](#) a close cousin — teaching simulators for a teaching-friendly ISA, with the same emphasis on observability and single-stepping — and the gem5 simulator [\[12\]](#) the production-research end

of the spectrum, where cycle-accurate timing models replace our one-instruction-equals-one-cycle abstraction. The distinction between *simulation* (modelling a machine's behaviour) and *emulation* (faithfully reproducing it, timing and all), and the technique of deterministic replay that our reproducible traces are a simple instance of, are treated thoroughly in Smith and Nair's survey of virtual machines [76].

Finally, the empirical observation that memory traffic dominates the instruction mix is not peculiar to our sieve; it is one of the oldest and most durable findings in computer architecture, and the reason the field has spent fifty years building cache hierarchies to make loads and stores cheaper. Hennessy and Patterson [35] is the standard account, and its opening chapters are the natural next step for a reader who, having watched `LOAD` consume a quarter of the sieve's cycles, wants to know what real hardware does about it.

Part IV

Beyond Compilation

Chapter 11

Interpreting instead of compiling

“A language that doesn’t affect the way you think about programming, is not worth knowing.”
— Alan J. Perlis

Every chapter so far has added another link to a chain that turns C into something else. The lexer turned characters into tokens, the parser turned tokens into a tree, the semantic analyser decorated that tree with types, the lowering walker flattened it into a typed intermediate representation, the optimiser rewrote that IR into a leaner version of itself, and the code generator turned the IR into FRISC assembly that the simulator of [Chapter 10](#) finally ran — all 44,355 cycles of the prime sieve we drove through it there — to print the one number the program existed to compute. Every stage was a translation. The program we wrote in [Chapter 6](#) did not *run* anywhere along the way; it was reshaped, again and again, until it reached a form some other machine knew how to execute. The whole pipeline is, in the end, an elaborate apparatus for handing the work off to FRISC.

Stop the pipeline at the IR and a quietly subversive question opens up. We already have a typed IR — a small, regular, fully-specified language of three-address instructions and basic blocks. What if we just *ran* it? Not lowered it, not translated it, but read each instruction and did what it says, on the spot, in Java, with no FRISC in sight. The IR is simple enough that this is not a fantasy: every instruction is one of three shapes, every right-hand side is one of ten, every value is a temporary or a constant. A program that walks that structure and carries out each operation is an *interpreter*, and writing one turns out to be both shorter than the code generator and, in a precise sense we will come to, the most honest description of what the IR *means*.

Perlis was writing about Lisp, and about the way a language can rearrange your sense of what programming is. The rearrangement we are after is smaller but real: once you have seen that the same IR can be *compiled* to FRISC and *interpreted* directly in Java, and that the two agree on every program, the IR stops being a way-station on the road to assembly and becomes a thing in its own right — a small machine with two implementations. By the end of the chapter we will have run the Fibonacci anchor without generating a single FRISC instruction, watched it return 6765 from the interpreter, and measured what that convenience costs.

11.1 Two ways to make a program run

There are, broadly, two ways to get a program to do something. You can *compile* it: translate it ahead of time into the instruction set of some machine, and then let that machine run the translation. Or you can *interpret* it: keep the program in some structured form and walk that structure with another program that performs each operation as it encounters it. The first approach pays a translation cost once, up front, and then runs at the speed of the target machine. The second pays no translation cost but re-examines the program's structure every single time it executes an instruction. Everything else — speed, portability, start-up latency, the ease of building the thing in the first place — follows from that one difference about *when* the work of understanding the program happens [67].

FRISCCc, up to this chapter, has been a pure compiler. It does all of its understanding ahead of time and emits FRISC that carries none of the source structure with it; by the time the simulator runs `a.out`, the notion of “a for loop” or “a local named a” has been ground away into registers, labels, and word offsets. That is exactly what we want from a compiler, and it is why the generated code is fast. But it also means the only way to find out what an IR program *does* has been to push it all the way through codegen and simulation. The IR itself has been mute — a specification we trusted rather than a thing we could question.

An interpreter changes that. The position we are in is unusually comfortable, because the hard part of interpretation — deciding what the program structure *is* — has already been done. A classic interpreter in the mould of the early Lisp and BASIC systems walks the abstract syntax tree directly [67, 72]: it descends into an `if` node, evaluates the condition subtree, and then walks one branch or the other. That works, but the AST is an awkward thing to interpret. It still carries the full irregularity of the source language — operator precedence, nested expressions, the difference between a statement and an expression — and the interpreter has to re-derive, at every node, facts the semantic analyser already established once. We are not going to interpret the AST. We are going to interpret the IR, and the IR is a far better target precisely because so much has already been decided about it.

The interpretation–compilation spectrum is not a binary. The two-way split is a useful first cut, but real systems live on a spectrum. A pure tree-walker sits at one end; a pure ahead-of-time compiler at the other. In between are bytecode interpreters (compile to a compact virtual instruction set, then interpret *that* — the subject of Chapter 12), just-in-time compilers (interpret at first, compile the hot paths to native code while the program runs), and threaded-code systems. The question is never “interpret or compile?” but “how much translation, and when?” FRISCCc's IR interpreter is deliberately at the tree-walking end: no translation at all beyond the lowering we already do, every instruction re-examined on every execution. It is the slowest design on the spectrum and the simplest, and for a teaching compiler that is

exactly the right trade.

The IR has three properties that make it the sweet spot for a direct interpreter. It is *already typed*: every instruction was checked in [Chapter 5](#) and carries its result type, so the interpreter never has to ask whether + means integer or fixed-point addition — the IR already says. It is *already lowered*: there are no nested expressions, no precedence, no implicit control flow; an if statement is two basic blocks and a conditional branch, and the interpreter just follows the branch. And it is *still target-independent*: the IR has no registers, no spill slots, no FRISC instruction selection, so an interpreter for it is small and the same IR runs unchanged whether we later compile it for FRISC or for anything else. The AST has the first property but not the second; FRISC assembly has the second but not the third. The IR is the only representation in the pipeline that has all three at once.

11.2 Walking the IR, not the tree

The interpreter is a single Java class, `IrInterpreter`, in the CLI module. Its constructor takes exactly one structural argument: an `IrProgramModel` — the same type the FRISC code generator consumes back in [Chapter 8](#). Two completely different back ends, the compiler and the interpreter, read the identical typed IR model. We will make a great deal of that fact in [Section 11.7](#); for now, notice only that the interpreter is not a re-implementation of the language but a second *reader* of the one IR we already produce.

Interpreting the IR means, at bottom, a cascade of decisions: given a piece of IR, which kind of piece is it, and what do we do for that kind? The IR model from [Chapter 6](#) is built from sealed interfaces — `Instruction` permits exactly `Assign`, `Store`, and `VoidCall`; `Rhs` permits exactly ten shapes; `Value` permits `Temp` and `Const`; `Terminator` permits `Br`, `Jmp`, and `Ret`. Each sealed interface is a closed menu, and the interpreter has one handler per menu item. [Figure 11.1](#) shows the whole cascade: `executeMain` calls `executeFunction`, which walks the blocks of a function; each block runs its instructions through `executeInstruction` and then dispatches its terminator; an `Assign`'s right-hand side flows through `evaluateRhs`, whose ten cases bottom out in `evaluateValue`, which resolves a temporary or a constant to a plain integer.

Reading the figure top to bottom is reading the interpreter's source top to bottom. There is no clever indirection, no jump table, no threaded dispatch — just a chain of `instanceof` pattern matches, one per sealed subtype, each doing the obvious thing. The virtue of the sealed-interface model surfaces here: because each interface is a closed set, the compiler can tell us if we forget a case, and a reader can be certain the cascade is exhaustive. The IR defines the menu; the interpreter orders one of everything.

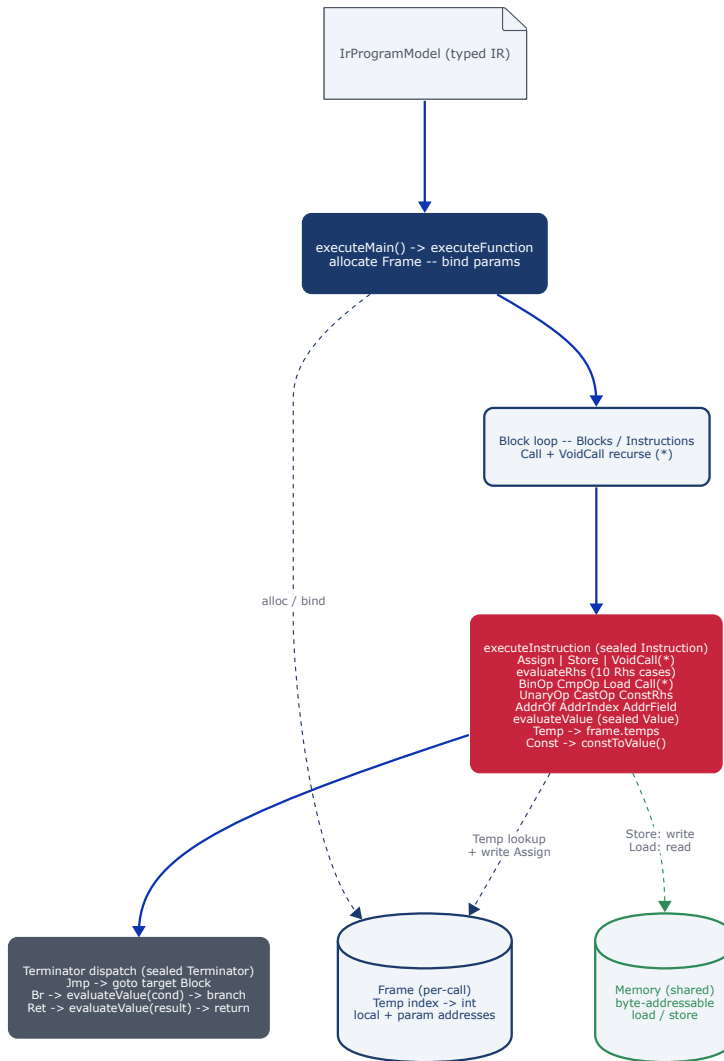


Figure 11.1: The dispatch structure of `IrInterpreter`. The same `IrProgramModel` the FRISC code generator consumes enters at the top. `executeFunction` walks each block, runs its instructions through `executeInstruction`, and ends each block by dispatching its terminator. An assignment's right-hand side fans out through `evaluateRhs` into the ten Rhs cases, all of which bottom out in `evaluateValue`. Two pieces of state sit to the side: a per-call `Frame` (temporaries and slot addresses) and a single shared byte-addressable `Memory`. The starred edge marks the recursion: a `Call` or `VoidCall` re-enters `executeFunction`.

Sealed types make the dispatch honest. The lowering walker, the optimiser, the code generator, and now the interpreter all dispatch over the same sealed hierarchy with the same instance of chains [11]. Sealing the interfaces — sealed interface `Rhs` permits `AddrOfSymbol`, `AddrIndex`, `AddrField`, `Load`, `BinOp`, `CmpOp`, `Call`, `UnaryOp`, `CastOp`, `ConstRhs` — turns “did I handle every shape?” from a question of discipline into a question the Java compiler answers. When we add a new `Rhs` in a later refactor, every dispatch site that does not yet handle it becomes a compile error, and the interpreter is one of those sites. The closed menu is what lets four independent consumers of the IR stay in agreement about what the IR can be.

With the dispatch settled — and the Java compiler standing guard over its completeness — the interpreter still needs somewhere to put the values it computes as it runs. That is the next piece to build.

11.3 An abstract machine: frames and byte-addressable memory

When the interpreter starts the Fibonacci function, the very first thing it does is carve out five four-byte slots in a row, beginning at address 4096: `n` at 4096, `a` at 4100, `b` at 4104, `i` at 4108, `t` at 4112. The values it computes as it runs — `t0 = 4096`, `t5 = 0`, `t8 = 1` — live somewhere else again. Those two somewheres are the interpreter’s entire runtime state: a `Memory` that holds the slots, and a `Frame` that holds the temporaries. Between them they model, in two small data structures, exactly the machine the IR was written against — everything in a single currency, the 32-bit word, and everything reached through a single idiom, the byte address.

The first is the `Memory`: a flat map from integer address to byte. It allocates upward from 0x1000 (4096 in decimal), bumping a pointer as each global, slot, and array asks for space, and aligning each request to its natural boundary. There is no stack and heap distinction, no protection, no virtual addressing — just bytes indexed by integers, exactly the model the FRISC code generator targets and the simulator implements. A 32-bit word is stored as four little-endian bytes, low byte first, so writing the value 20 to address 4096 sets byte 4096 to 20 and bytes 4097 through 4099 to zero. `storeWord` and `loadWord` do the byte arithmetic; everything above them deals in words and addresses and never thinks about endianness again.

The second is the `Frame`: the activation record for one function call. It holds the addresses of the function’s parameter and local slots (each allocated in `Memory` when the call begins), the types of those slots, and — this is the part with no analogue in Chapter 6 — a map from temporary index to its current integer value. Temporaries in the IR are not memory; they are the wires between instructions, and in the interpreter they live in a plain `Map<Integer, Integer>` on the frame, created fresh for every call and discarded when the call returns.

Figure 11.2 draws both structures for the Fibonacci anchor, caught one iteration into the loop.

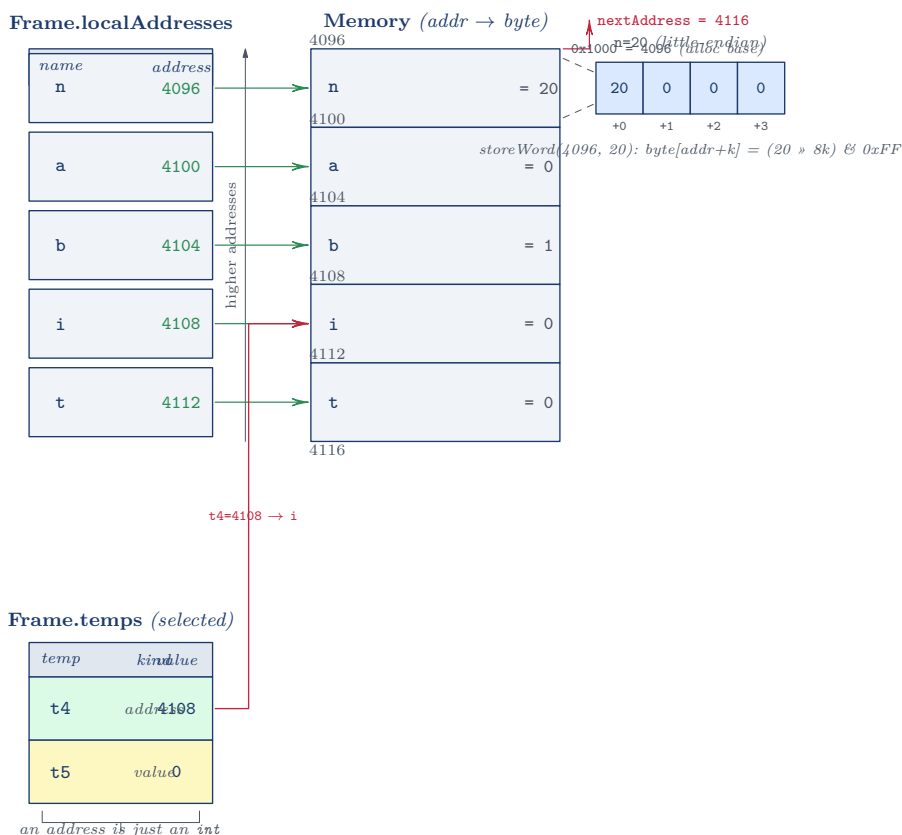


Figure 11.2: The `IrInterpreter` runtime state for `math_fibonacci_iter`, captured at the first entry to the loop-condition block. Memory is a flat map from integer address to byte, allocated upward from `0x1000`; the five `int32` locals occupy consecutive four-byte slots, and the inset explodes `n = 20` into the little-endian bytes `[20, 0, 0, 0]`. The Frame holds two maps: `localAddresses`, mapping each name to its slot address (green arrows), and `temps`, mapping each temporary index to its current value. Temporary `t4 = 4108` is an *address* — it points at slot `i` — while `t5 = 0` is a loaded *value*. Both are stored in the same integer-to-integer map; the interpreter never distinguishes them.

That last observation is the conceptual heart of the section, and it is worth stating plainly.

Key insight

An address is just an integer. The IR has no run-time pointer type that differs from an `int32`; `addr_of_symbol` produces a value, `load` consumes a value as an address, and

arithmetic on addresses (`addr_index`) is ordinary integer arithmetic. The interpreter takes this literally. A temporary holding the *address* of `i` and a temporary holding the *value* loaded from `i` are both `Integers` in the same map. This is precisely the untyped, byte-addressable view of memory that C exposes and that FRISC implements, and reproducing it faithfully is what lets the interpreter agree with the compiled program down to the last load.

This single-currency model is also why the interpreter can run programs that use arrays, structs, and pointers without any of those becoming special cases in the dispatch. An array is a run of bytes; a struct field is a fixed offset added to a base address; a pointer is a word holding an address. The instruction `addr_index` computes `base + index * elemSize` — plain integer arithmetic — and `addr_field` adds the field’s compile-time offset to a base. Aggregates that are passed or assigned whole are copied byte-for-byte with a `memcpy`-style loop. None of this needs a richer notion of value than “a 32-bit integer that might be a number or might be an address,” which is the only notion FRISC has either.

Fixed-point arithmetic, reproduced exactly. FRISCcc has no floating-point hardware; it represents `float` in the Q16.16 fixed-point format, a 32-bit integer holding sixteen bits of integer part and sixteen of fraction. The interpreter must reproduce that arithmetic to the bit, or it would disagree with the compiled program on every floating-point operation. So a `float` *is* stored as its Q16.16 integer encoding, conversion from `int` to `float` is a left shift by sixteen (`itof`), the reverse is an arithmetic right shift (`ftoi`), multiplication computes `(long) a * b >> 16` to avoid overflow in the intermediate product, and division computes `((long) a << 16) / b`. These are the same routines the runtime helpers `F_Fmul` and `F_Fdiv` implement on FRISC (Chapter 9); the interpreter simply inlines them. Reproducing the target’s arithmetic quirks, rather than borrowing the host’s, is the difference between an interpreter that *models* the machine and one that merely resembles it.

`Memory` and `Frame` are the whole of the interpreter’s abstract machine. With them in hand we can write the engine that drives it — the loop that turns a control-flow graph into a run of executed instructions.

11.4 The interpreter loop

With the state in place, the engine is short. Running a function means walking its basic blocks. The interpreter keeps a “current block” pointer, runs every instruction in that block in order, and then looks at the block’s terminator to decide which block becomes current next. A `Jmp` names its successor directly; a `Br` evaluates its condition and picks one of two successors; a `Ret` ends the walk and produces the function’s return value. There is no program counter threading through a flat instruction array, because the IR is not flat — it is a graph of blocks, and the interpreter walks the graph. It is the same pointer-chasing a

depth-first search does over a directed graph: hold a reference to the current node, process its contents, then let the outgoing edges — here, the block’s terminator — choose which node to visit next. [Algorithm 11.1](#) states the loop.

Algorithm 11.1: The block-walking interpreter loop. The current block runs its instructions in order, then its terminator chooses the next block; **Ret** ends the walk. Control flow is graph traversal, not a program counter.

Input: a function f and a list of argument values $args$

Output: the integer value returned by f

```

1   $F \leftarrow$  new Frame
2  allocateSlots( $f$ ,  $F$ )                (reserve memory for every param/local slot)
3
4  bindParameters( $f$ ,  $args$ ,  $F$ )        (copy arguments into param slots)
5
6   $cur \leftarrow$  first block of  $f$ 
7  while  $cur \neq \text{nil}$  do
8      for each instruction  $I$  in  $cur$  do
9          executeInstruction( $f$ ,  $cur$ ,  $I$ ,  $F$ )    (one tick per instruction)
10
11      $T \leftarrow$  terminator of  $cur$ 
12     switch kind of  $T$  do
13         case Ret  $v$  do
14             if  $v = \text{nil}$  then return 0
15             return evaluateValue( $v$ ,  $F$ )
16         case Jmp  $L$  do
17              $cur \leftarrow$  block labelled  $L$ 
18         case Br  $c, L_t, L_f$  do
19             if evaluateValue( $c$ ,  $F$ )  $\neq 0$  then  $cur \leftarrow$  block  $L_t$ 
20             else  $cur \leftarrow$  block  $L_f$ 
21 return 0

```

The loop makes the IR’s control-flow graph concrete. For the Fibonacci anchor, the graph has five blocks — [Figure 11.3](#) — and the interpreter threads through them exactly as the conditional branches direct: an entry block that initialises the variables, a condition block reached once per iteration, a body block, an increment block that jumps back to the condition, and an exit block whose terminator is the **ret** that ends the walk. The back-edge from the increment block to the condition block is the loop; the interpreter follows it the same way it follows any other **jmp**, which is why a loop costs nothing special to interpret beyond running its blocks again.

Running an instruction is the second half of the engine. Of the three instruction shapes, **Store** writes a value (or copies an aggregate) to a computed address, **VoidCall** invokes a function for its effect and discards the result, and **Assign** computes a right-hand side and

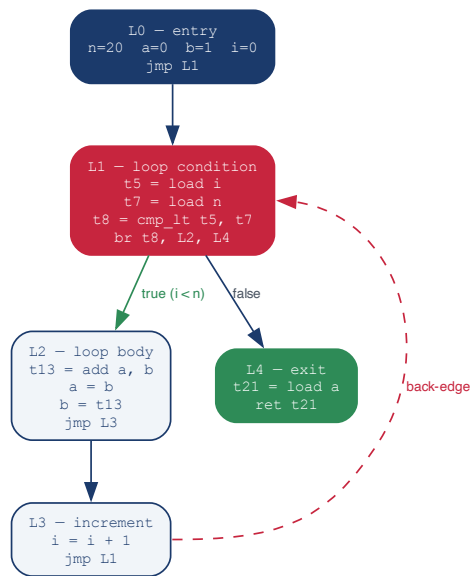


Figure 11.3: The control-flow graph of the Fibonacci anchor’s unoptimised IR, and the path the interpreter walks. `L0` initialises; `L1` tests `i < n` and branches to the body (`L2`) or the exit (`L4`); `L2` computes one Fibonacci step; `L3` increments `i` and takes the dashed back-edge to `L1`; `L4` returns `a`. The interpreter follows the back-edge twenty times before the condition fails.

binds the resulting integer to a destination temporary in the frame. The interesting work is in evaluating the right-hand side, because that is where the ten Rhs shapes turn into actual arithmetic, memory access, and calls. Algorithm 11.2 walks the cascade.

Algorithm 11.2: Evaluating an Rhs to an integer. Every case bottoms out in `evaluateValue` on its operands, which resolves a temporary against the frame’s map or decodes a constant. The `Call` case re-enters Algorithm 11.1, which is where the call stack lives.

Input: a right-hand side r and the current frame F

Output: the integer value of r

```

1 switch kind of r do
2   case ConstRhs k do
3     return integer encoding of  $k$ 
4   case AddrOfSymbol s do
5     return address of slot  $s$  in  $F$ 
6   case AddrIndex b, x, sz do
7     return evaluateValue(b) + evaluateValue(x) × sz
8   case AddrField b, fld do
9     return evaluateValue(b) + offset of fld
10  case Load a : τ do
11    return loadWord(evaluateValue(a))           (one byte if  $\tau$  is char)
12  case BinOp ⊙, l, r' do
13    return apply ⊙ to evaluateValue(l), evaluateValue(r')   (integer or Q16.16)
14  case CmpOp ~, l, r' do
15    return 1 if evaluateValue(l) ~ evaluateValue(r'), else 0
16  case UnaryOp ⊖, o do
17    return apply ⊖ to evaluateValue(o)
18  case CastOp κ, o do
19    return cast κ of evaluateValue(o)
20  case Call g, a1 ... ak do
21    return executeFunction(g) applied to evaluateValue(a1), ..., evaluateValue(ak)

```

`evaluateValue`, at the bottom of every case, is the simplest routine in the interpreter: a Temp is looked up in the frame’s temporary map — and it is a hard error if it has not been assigned, though the IR’s definition-before-use invariant from Chapter 6 guarantees it always has — and a Const is decoded to its integer encoding. Everything the interpreter does is some composition of these two leaves with the arithmetic and memory operations in between. The `BinOp` case is where integer and fixed-point arithmetic diverge: the result type carried on the instruction — checked once, back in semantic analysis — tells the interpreter whether `mul` means a plain product or the Q16.16 routine from the sidebar above, so the interpreter never has to guess.

11.5 Calls, returns, and the host's call stack

A function call in [Algorithm 11.2](#) is one line: the `Call` case evaluates the argument values and re-enters `executeFunction`. That single line hides the whole calling convention, and it hides it by borrowing one. When `executeFunction` calls itself, the new invocation gets its own Java stack frame, its own local `Frame` object, its own temporary map. The IR program's call stack *is* the Java call stack; the interpreter does not maintain an explicit stack of activation records at all. A recursive IR program becomes a recursive Java program, and the host runtime does the bookkeeping.

[Algorithm 11.3](#) spells out the protocol that `executeFunction` runs at the top of every call — the slot allocation and parameter binding that [Algorithm 11.1](#) abbreviated. Each slot the function declares gets fresh memory, zeroed; each parameter's argument value is written into its slot, or, for an aggregate, the bytes are copied from the caller's address. When the function returns, the frame object goes out of scope and its temporaries vanish, but — and this is the subtlety of a byte-addressable model — the *slots* it allocated in Memory are not reclaimed. The allocator only ever bumps upward. For the terminating programs in our test suite this never matters; we return to it when we talk about limits.

Algorithm 11.3: The slot allocation and parameter binding run at the start of every call. Scalars are passed by value; aggregates are passed by copying their bytes. The call stack is the host language's — `executeFunction` recurses, and Java's own stack records the chain of activations.

Input: a function f , argument values $args$, a fresh frame F

Output: F populated with slot addresses and bound parameters

```

1 allocateSlots:
2 for each slot  $s$  in  $f$  do
3    $a \leftarrow \text{alloc}(\text{sizeof}(s.\text{type}), \text{alignOf}(s.\text{type}))$ 
4   record  $a$  as the address of  $s$  in  $F$ , by kind (param or local)
5   zero the  $\text{sizeof}(s.\text{type})$  bytes at  $a$ 
6 bindParameters:
7 for  $i \leftarrow 0$  to  $|args| - 1$  do
8    $p \leftarrow$  the  $i$ -th parameter of  $f$ ;  $a \leftarrow$  address of  $p$ 's slot
9   if  $p.\text{type}$  is an aggregate then
10    copy  $\text{sizeof}(p.\text{type})$  bytes from  $args[i]$  to  $a$  (by reference)
11  else
12     $\text{storeWord}(a, args[i])$  (scalar by value)
13  end for
14
```

Leaning on the host's call stack is a venerable shortcut. It is the same move that makes a *meta-circular* interpreter — an interpreter for a language written in that same language — so short and so illuminating [67, 72]. We get recursion, re-entrancy, and correct nesting for

free, because the host language already solved those problems and we are standing on its solution. The price is that we inherit the host's limits as well as its conveniences, and the most visible limit is depth.

The shortcut has a floor. Because each IR call becomes a Java call, a deeply recursive IR program can exhaust the Java thread's stack and raise a `StackOverflowError` long before it would have run out of FRISC stack at address 40000. The two machines have different stack limits, so a program that overflows on one might not on the other. This is the first place the interpreter and the compiled program can *legitimately* disagree — not on the value a program computes, but on whether a pathologically deep recursion completes at all. The bytecode VM of [Chapter 12](#) takes the opposite design: it maintains its own explicit stack of frames, pays for the bookkeeping the interpreter got for free, and in exchange controls its own depth limit independently of the host. The contrast between the two designs — borrow the host's stack, or build your own — is one of the main reasons the next chapter exists.

Stack depth is one way the interpreter must answer for the machine it models. The others are sharper, because they are the points where a program is supposed to *fail* — and an interpreter that gets the successes right but the failures wrong is not faithful at all.

11.6 Trapping what FRISC traps

An interpreter that only ran correct programs would be easy. The harder obligation is to fail where the compiled program fails, and for the same reasons, so that the interpreter is a faithful model of the target's behaviour and not just of its happy path. FRISCcc's runtime ([Chapter 9](#)) traps three kinds of fault — an out-of-bounds array access, a division by zero, and a runaway program — and the interpreter reproduces each.

Array bounds are the most interesting, because the interpreter does a small amount of analysis to decide *which* address computations to check. An `addr_index` that merely computes a pointer for arithmetic need not be checked; one whose result feeds a load or store must be, because that is where an out-of-range index turns into a wild memory access. So before running a function the interpreter walks its instructions and computes, to a fixed point, the set of temporaries whose values flow into a memory address. An `addr_index` whose destination is in that set is bounds-checked at run time against the array's declared size; one whose destination is not is left alone. When a checked index falls outside its array, the interpreter raises a trap carrying code -6 — the same code the `L_BOUNDS_ERROR` trap reports on FRISC — and that becomes the program's exit value, exactly as it would after compilation.

A miniature dataflow analysis, inside the interpreter. The bounds-check decision is a backward reachability question: starting from the address operands of every load and store, which temporaries can flow into an address? The interpreter seeds the set with those operands and then iterates — if an `addr_field` or `addr_index` produces a temporary already in the set, its base is added too — until the set stops growing. This is the same fixed-point shape as the worklist analyses of [Chapter 7](#), in miniature, and it is a quiet reminder that the boundary between “analysis” and “execution” is porous: even a direct interpreter benefits from looking at the program before it runs it.

The other two faults are simpler. Integer division or modulo by zero returns zero rather than throwing, matching the defensive convention of the FRISC division helpers `F_DIV` and `F_MOD`, so a divide-by-zero never crashes the interpreter or the simulated program; both produce the same defined-but-meaningless result. And every instruction and terminator the interpreter executes ticks a step counter against a watchdog limit, two million steps by default. A program that exceeds it — an infinite loop, or simply one larger than the interpreter was meant to run — is stopped with a clear diagnostic rather than hanging the tool. The watchdog has no analogue in the compiled program, which would loop forever on real hardware; it is a concession to the fact that an interpreter runs inside a development tool that ought not to wedge.

11.7 How fast is fast enough?

We can now ask what interpretation costs. The natural unit is the interpreter’s own *step*: one instruction or one terminator executed, counted by the watchdog. It is not wall-clock time — that depends on the JVM, the machine, and the weather — but it is a clean, reproducible measure of how much work the interpreter does, and it maps directly onto the dispatch cascade of [Figure 11.1](#): every step is one trip down that cascade. [Figure 11.4](#) plots the step count for a spread of programs from the test suite.

Two things stand out. The first is that the step count tracks the program’s actual workload, not its source size: the eight-line `gcd` routine of `math_gcd_lcm` runs to completion in 72 steps because Euclid’s algorithm converges fast, while the prime sieve runs 10 587 because it really does touch every cell of its array many times. The interpreter has no fixed overhead to amortise; it pays per instruction executed, so its cost is the dynamic instruction count, full stop. The second is that optimisation helps the interpreter for exactly the reason it helps the compiled program. The optimiser does not know or care that an interpreter will run its output — it removes redundant loads, folds constants, and hoists invariants because those make *any* execution shorter. Feeding the optimised Fibonacci IR to the interpreter cuts the step count from 478 to 378 because there are literally a hundred fewer instructions and terminators to dispatch over the program’s run.

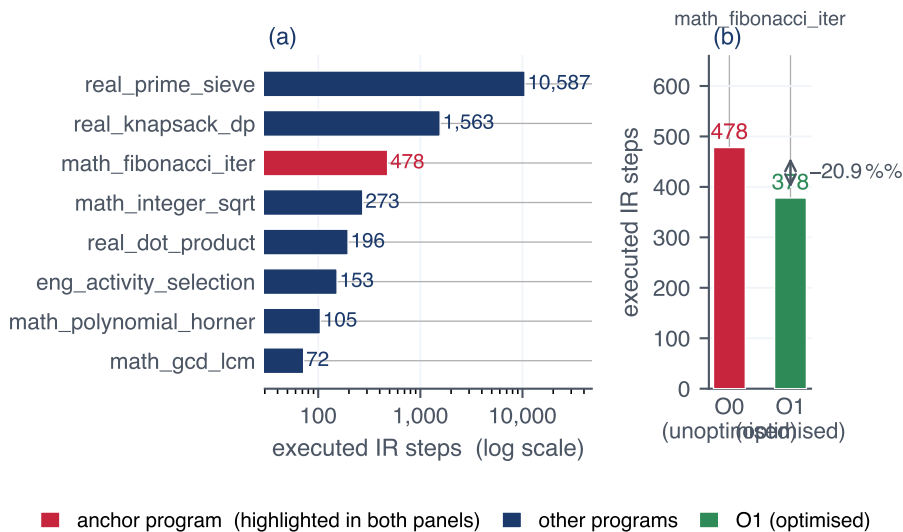


Figure 11.4: Left: executed IR steps for eight programs from the test suite, on a logarithmic axis; the anchor `math_fibonacci_iter` is highlighted. The range spans two orders of magnitude, from a 72-step gcd to a 10 587-step prime sieve, tracking the amount of computation each program actually performs. Right: the effect of optimisation on the anchor — the unoptimised IR takes 478 steps, the optimised IR 378, a 20.9% reduction — and both still return 6765.

How fast is fast enough, then? For its purpose, the interpreter is plenty fast: it runs every program in the test suite in well under a second, which is all a development tool needs. It would be a terrible way to run a long-lived production workload — re-examining the program structure on every instruction is exactly the cost a compiler exists to pay once and avoid forever — and that gap, the interpreter’s per-instruction tax against the compiler’s amortised-to-zero translation, is the whole economic argument for compilation. But speed was never the reason to build this interpreter. The reason is in the next box.

Key insight

The interpreter is a differential oracle for the compiler. Because `IrInterpreter` and the FRISC code generator consume the *same* `IrProgramModel`, we have two independent implementations of the IR’s semantics: one that runs it directly in Java, one that translates it to FRISC and runs the translation on the simulator. Run a program both ways. If they disagree on the result, exactly one of two things is wrong — the code generator mistranslated the IR, or the interpreter misread it — and the disagreement is a test failure pointing straight at the bug. This is *differential testing* [60], and it is among the most powerful debugging tools in the whole project: every example program is automatically a cross-check between two back ends that share nothing but the IR they read. The interpreter pays for itself the first time it catches a codegen bug the simulator alone would have let through.

There is a second, quieter payoff. An interpreter written this directly is the clearest possible specification of what the IR *means*. The formal way to say “what an instruction does” is an operational semantics — a set of inference rules relating a machine state before an instruction to the state after [42, 70, 84]. `evaluateRhs` and `executeFunction` are those rules, written in Java and made to run. When [Appendix G](#) proves an optimisation pass preserves semantics, “semantics” is, concretely, the behaviour this interpreter exhibits. The interpreter is where the IR’s meaning stops being a promise and becomes something we can execute and check.

11.8 Watching the anchor run

Let us run the Fibonacci anchor through the interpreter and watch it work. The source ([Listing 7](#)) is the iterative twenty-line program we have carried since [Chapter 3](#); its unoptimised IR is the five-block graph of [Figure 11.3](#). We invoke the interpreter directly on the IR, bypassing codegen entirely:

```
1 | ./run.sh run-ir book/listings/math_fibonacci_iter/intermediate_00.ir
```

which prints `Return value: 6765` and `Executed steps: 478`. The right answer, and a precise count of the work. Asking for a trace with `--trace-ir` prints one line per dispatch; [Listing 8](#)

shows the opening of that trace — the entry block running to completion and the first pass through the condition and body blocks.

```

1  // EXPECT: 6765
2
3  int main(void) {
4      int n = 20;
5      int a = 0;
6      int b = 1;
7      int i;
8      int t;
9
10     for (i = 0; i < n; i++) {
11         t = a + b;
12         a = b;
13         b = t;
14     }
15
16     return a;
17 }
```

Listing 7: The anchor: an iterative Fibonacci that returns 6765 for $n = 20$, carried since Chapter 3 and now run directly from its IR.

The trace is the dispatch cascade made visible, one line at a time. In block L0, step 1 evaluates `addr_of_symbol local:n` — `t0 = 4096`, the address of the first slot — and step 2 runs the `store` that writes 20 there. The interpreter walks the four initialisations this way and then, at step 9, dispatches the `jmp` terminator into L1. There the loads materialise as `t5 = 0` (the current value of `i`) and `t7 = 20` (the value of `n`), the comparison `cmp_lt` produces `t8 = 1` because zero is less than twenty, and the `br` at step 15 takes the true edge into the loop body. Notice the two faces of a temporary on display: `t4 = 4108` is an address, the location of `i`; `t5 = 0` is the value found there. The interpreter stores both as plain integers, exactly as Figure 11.2 promised.

From there the interpreter simply keeps walking. It rounds the loop twenty times, following the back-edge from the increment block to the condition block each time, until `i` reaches 20 and the comparison finally yields false. The trace ends in block L4, where step 477 loads the final value of `a` — `t21 = 6765` — and the `ret` terminator at step 478 ends the walk and hands that value back. No registers were allocated, no FRISC instruction was selected, no `a.out` was written. We took the same typed IR the compiler would have lowered to assembly and instead read it directly, and it told us, in 478 steps, that the twentieth Fibonacci number is 6765 — the very answer the compiled program produces on the simulator. Two back ends, one IR, one result.

Practice

```

1    [1] main:L0 -> instr[0]
2    [1] main:L0 -> t0 = 4096
3    [2] main:L0 -> instr[1]
4    [2] main:L0 -> store[4096] = 20
5    [3] main:L0 -> instr[2]
6    [3] main:L0 -> t1 = 4100
7    [4] main:L0 -> instr[3]
8    [4] main:L0 -> store[4100] = 0
9    [5] main:L0 -> instr[4]
10   [5] main:L0 -> t2 = 4104
11   [6] main:L0 -> instr[5]
12   [6] main:L0 -> store[4104] = 1
13   [7] main:L0 -> instr[6]
14   [7] main:L0 -> t3 = 4108
15   [8] main:L0 -> instr[7]
16   [8] main:L0 -> store[4108] = 0
17   [9] main:L0 -> terminator
18   [10] main:L1 -> instr[0]
19   [10] main:L1 -> t4 = 4108
20   [11] main:L1 -> instr[1]
21   [11] main:L1 -> t5 = 0
22   [12] main:L1 -> instr[2]
23   [12] main:L1 -> t6 = 4096
24   [13] main:L1 -> instr[3]
25   [13] main:L1 -> t7 = 20
26   [14] main:L1 -> instr[4]
27   [14] main:L1 -> t8 = 1
28   [15] main:L1 -> terminator
29   [16] main:L2 -> instr[0]
30   [16] main:L2 -> t9 = 4100

```

Listing 8: The first thirty lines of the interpreter’s trace of the anchor. Each step prints twice: a dispatch line naming the instruction (`instr[k]` or `terminator`) and an effect line showing what it did. In block `L0` the interpreter assigns the address of each slot to a temporary and stores the initial value; `store[4096] = 20` writes `n`. Block `L1` loads `i` and `n` into `t5` and `t7` and computes the comparison into `t8 = 1`, taking the branch into the body.

Exercise 11.1 (Applied · ★★★)

Counting steps by hand. Using the control-flow graph in [Figure 11.3](#) and the unoptimised IR for the anchor, work out the executed-step count *from first principles*. Block L0 has eight instructions and one terminator; L1 has five instructions and one terminator and runs once per loop test; L2, L3, and L4 have their own counts. The loop test in L1 runs 21 times (twenty times taking the body, once failing), while the body L2 and increment L3 run 20 times each. Add up the instructions and terminators executed across the whole run and confirm that you get 478, the figure the interpreter reports. Which block contributes the most steps, and why?

Exercise 11.2 (Applied · ★★★)

Why optimisation saved exactly a hundred steps. The optimised IR runs the anchor in 378 steps, 100 fewer than the unoptimised 478. Open both IR dumps (intermediate_00.ir and intermediate_01.ir in the anchor’s listing directory) and compare the loop body. Note first that at -01 the optimiser merges the increment block L3 into L2, so the optimised dump has no separate L3: the comparison is between 00’s two-block body (L2 at twelve steps plus L3 at five, seventeen steps per iteration) and 01’s single-block body (twelve steps per iteration). Count the per-iteration saving, multiply by the number of times the body runs, and account for the difference. Why does shaving five steps from a loop body save a hundred steps overall rather than five?

Exercise 11.3 (Applied · ★★★)

Tracing a temporary’s two lives. In the trace excerpt of [Listing 8](#), temporary t4 holds the value 4108 and temporary t5 holds 0. Explain, referring to the IR for block L1, why one of these is an address and the other a value, and which IR instruction produced each. Then answer: when the interpreter later runs store t9, t15 in the loop body, how does it know to interpret t9 as a destination address rather than as a number to be stored? What information on the Store instruction settles the question?

Exercise 11.4 (Applied · ★★★)

The bounds-check analysis. [Section 11.6](#) describes the fixed-point analysis that decides which `addr_index` computations to bounds-check. Consider an IR fragment that computes `t1 = addr_index t0, ti, 4`, then `t2 = load t1 : int32`, and separately computes `t3 = addr_index t0, tj, 4` whose result is never loaded from or stored to — it is only compared against another pointer. Walk the analysis: which of t1 and t3 ends up in the “flows-into-an-address” set, and therefore which `addr_index` is checked at run time? Why is it safe — and not merely an optimisation — to skip the check on the unchecked one?

Exercise 11.5 (Applied · ★★★)

Where the two back ends may legitimately differ. The chapter claims the interpreter and the compiled program agree on the *value* every terminating program computes, but may disagree on whether a deeply recursive program completes at all. Explain the mechanism: what resource does the interpreter exhaust that the compiled program does not, and vice versa? Then describe one program shape that would complete under one back end and fail under the other, and state which fails on which. Is this a bug in the interpreter, a bug in the compiler, or neither? Justify your answer in two or three sentences.

Project

Exercise 11.6 (Lab · ★★★)

Project: a tree-walking interpreter for tinyC's IR. Build a direct interpreter for the tinyC IR you generated in [Chapter 6](#)'s project, so that the same IR can be both compiled and interpreted — and the two cross-checked against each other.

Open `IrTreeWalker.java` in the companion repository's `ch11-treewalk/starter/` module, under the `com/frisccc/tinyc/interp/` package. The IR data model (`IrProgram`, `IrFunction`, `IrBlock`, the sealed `IrInstr`, `IrRhs`, `IrValue`, and `IrTerminator` hierarchies) is the one your lowering walker already produces; the runtime scaffolding — a byte-addressable `Memory` class, a `Frame` record, and the step-counting watchdog — is provided. tinyC is int-only, with no floats, arrays, structs, or pointers, so your interpreter is meaningfully smaller than FRISCCc's: there is no Q16.16 arithmetic, no aggregate copying, and no bounds-check analysis to write.

Four // TODO: implement markers await you, mirroring the four algorithms of this chapter:

1 TODO `interpretFunction`: allocate the frame, bind the arguments to parameter slots, and run the block-walking loop of [Algorithm 11.1](#) — run each instruction, then dispatch the terminator to choose the next block, stopping on `Ret`.

2 TODO `evalRhs`: the dispatch of [Algorithm 11.2](#), restricted to tinyC's shapes — constants, `addr_of_symbol`, `load`, the integer `BinOps` and `CmpOps`, `UnaryOp`, and `Call`.

3 TODO `execInstruction`: handle `Assign` (bind the evaluated right-hand side to the destination temporary), `Store` (write a value to a computed address), and `VoidCall`.

4 TODO division and modulo by zero: return zero rather than throwing, matching FRISCCc's helper convention so your interpreter and your eventual code generator agree.

The provided tests in `ch11-treewalk/tests/` run a battery of tinyC programs both

through your interpreter and through the reference FRISCcc-style compiled path, and assert the two return values *match* — the differential-oracle test of [Section 11.7](#) in miniature. A passing suite means your two back ends agree on every program. As a stretch goal, add a `--trace` flag that prints one line per dispatched instruction, in the style of [Listing 8](#), and use it to debug the first program whose two back ends disagree.

Notes and further reading

The clearest modern introduction to tree-walking interpreters, and the book whose narrative voice this one borrows, is Nystrom’s *Crafting Interpreters* [67], which builds both an AST-walking interpreter and a bytecode VM for the same language — precisely the two-chapter arc this part of the book follows. Queinnec’s *Lisp in Small Pieces* [72] is the deeper treatment of the design space, including the meta-circular interpreter and the long slide from naive tree-walking toward compilation; it is where the “how much translation, and when?” question of [Section 11.1](#) is dissected most carefully. The shortcut of leaning on the host’s call stack, and the broader habit of treating an interpreter and a compiler as two readings of one program, runs back through Steele’s early Lisp compilers [79] to the meta-circular tradition Queinnec documents.

The idea that an interpreter *is* an operational semantics — that `evaluateRhs` is a set of inference rules made executable — is made precise by Plotkin’s structural operational semantics [70] and Kahn’s natural (big-step) semantics [42]; Winskel’s *Formal Semantics of Programming Languages* [84] is the gentlest textbook on both. Our recursive `executeFunction` is a big-step evaluator in Kahn’s sense: it relates a call and its arguments directly to the returned value, without exposing the intermediate states a small-step semantics would. The connection matters for [Appendix G](#), where the “semantics” an optimisation pass must preserve is, operationally, the behaviour this interpreter exhibits.

Differential testing — running two implementations against each other and treating disagreement as a bug report — was named and popularised by McKeeman [60], and it is the practical reason the interpreter earns its place in the build. The general strategy of validating a compiler against a reference semantics for its IR reaches its most rigorous form in CompCert [56], where the reference semantics is not merely tested against but *proved* equivalent to the compiled code; our cross-check is the lightweight, unproved cousin of that idea. Readers who want the dispatch itself to go faster than a chain of `instanceof` tests should look at threaded code, and at Java’s own evolution toward pattern matching over sealed types [11], which is what keeps FRISCcc’s four IR consumers honest about handling every case.

The next chapter keeps the IR but stops walking its tree. We will compile the IR one step further — into a flat bytecode — and build a virtual machine with its own explicit operand stack and call stack, trading the tree-walker’s simplicity for the control over execution that [Section 11.5](#)’s sidebar said we would eventually want.

Chapter 12

A machine of our own

“All problems in computer science can be solved by another level of indirection.”
— David Wheeler

The interpreter of [Chapter 11](#) was a small triumph of doing less. We took the typed IR — the very same `IrProgramModel` the FRISC code generator consumes — and instead of translating it we simply read it, instruction by instruction, and did what each one said. It returned 6765 for the Fibonacci anchor without emitting a single FRISC instruction, and it agreed with the compiled program on every test we threw at it. It was shorter than the code generator and, in a precise sense, the most honest statement of what the IR means. But it had a quiet extravagance hidden inside its simplicity. Every time it executed an instruction, it re-examined the instruction’s *structure*: an `instanceof` test to find out which of ten right-hand-side shapes it was looking at, a hash-map probe to resolve each temporary, a pointer chased to the next basic block. The interpreter pays that cost of *understanding the program* afresh on every instruction, every time around every loop. A compiler pays it exactly once.

Pay that cost once and the inner loop is left with almost no thinking to do — which is the single step down the spectrum this chapter takes. We will not go all the way to FRISC — that is what [Chapter 8](#) already did. Instead we will compile the IR *one step further than the interpreter does*, into a flat *bytecode*: a linear stream of small numeric opcodes with their operands packed in beside them, no nested expressions, no symbolic block labels, no types. Then we will build a *virtual machine* to run that bytecode — a machine of our own design, with its own program counter threading through the byte stream, its own operand stack on which all arithmetic happens, and, most importantly, its own explicit stack of call frames. The sidebar at the end of [Section 11.5](#) promised this contrast: the tree-walker borrowed the host’s call stack and inherited its depth limit; the VM builds its own and controls that limit itself. Making good on that promise is one of the main reasons this chapter exists.

Wheeler’s quip is the whole idea in nine words. The bytecode is a level of indirection inserted between the IR and the act of running it, and like every good indirection it buys us something: the cost of understanding the program is paid once, at lowering time, so that the running loop has almost nothing left to understand. By the end of the chapter we will have lowered the prime-sieve anchor to bytecode, watched a virtual machine dispatch its way to 46, and measured exactly what the extra translation step costs and what it saves.

12.1 One more step down the spectrum

Recall the spectrum from [Section 11.1](#). At one end sits the pure tree-walker, which does no translation at all and re-examines the program structure on every instruction. At the other end sits the ahead-of-time compiler, which translates everything up front and leaves nothing to examine at run time. The question, we said, is never “interpret or compile?” but “how much translation, and when?”. The IR interpreter sits firmly at the tree-walking end. The bytecode VM we are about to build sits one notch toward the compiler: it translates the IR into a compact virtual instruction set, and then interprets *that*.

The move is worth making because the tree-walker’s per-instruction tax is not a fixed overhead — it is paid again and again. Consider a single line of the IR’s loop body, say `t8 = cmp_lt t5, t7`. Each time the interpreter reaches it, it runs an instance of `cascade` to discover that the right-hand side is a `CmpOp`, resolves `t5` and `t7` through a `HashMap`, and dispatches on the comparison operator. None of that work depends on the *values* in the temporaries; it depends only on the *shape* of the instruction, which never changes. In a loop that runs twenty thousand times, the interpreter rediscovers that shape twenty thousand times. The bytecode VM discovers it once, when it lowers the instruction, and records the answer as a handful of opcodes. At run time those opcodes say what to do directly — no searching, no shape analysis, just a small integer to switch on.

The middle of the spectrum is crowded. Between the tree-walker and the ahead-of-time compiler live most of the language implementations people actually use. The Java Virtual Machine and CPython compile to bytecode and interpret it; V8 and the HotSpot JVM start by interpreting bytecode and then *just-in-time* compile the hot paths to native code while the program runs [76]. The bytecode VM is not an exotic design — it is the workhorse of the field, the representation that makes a language portable (ship the bytecode, not the machine code) and fast enough (dispatch a small integer, do not walk a tree) at the same time. Our VM stops short of the just-in-time step, but it is built on exactly the substrate a JIT would start from. [Chapter 15](#) returns to what the next step up would change.

There is a second reason to build this, beyond speed, and it is the same reason the interpreter earned its place. The VM is a *third* independent reading of the IR’s meaning. We already had two — the FRISC code generator and the tree-walking interpreter — and any disagreement between them was a bug report. A third reader that shares nothing with the other two but the IR it consumes turns a pairwise check into a three-way triangulation. We will lean on that hard in [Section 12.6](#).

12.2 Designing the bytecode

What should the bytecode look like? The IR is a *three-address* language: every instruction names its operands and its destination explicitly, as in `t8 = cmp_lt t5, t7`. A bytecode could keep that shape, encoding three operand fields per instruction — that is a *register machine*, and it is the design [Chapter 15](#) sketches. We will take the other classic route, the one that makes for the smallest and simplest instructions: a *stack machine*.

On a stack machine, instructions do not name their operands. Instead they take them from the top of an *operand stack* and leave their result there. The instruction `cmp_lt` becomes a single opcode `CMP_LT` that pops two values and pushes a one or a zero; it does not need to say *which* two values, because they are, by construction, the top two on the stack. To get `t5` and `t7` onto the stack in the first place, we precede the comparison with two `LOAD_TEMP` opcodes, and to capture the result we follow it with a `STORE_TEMP`. So the one IR instruction `t8 = cmp_lt t5, t7` lowers to a little run of four stack instructions, shown in [Figure 12.1](#).

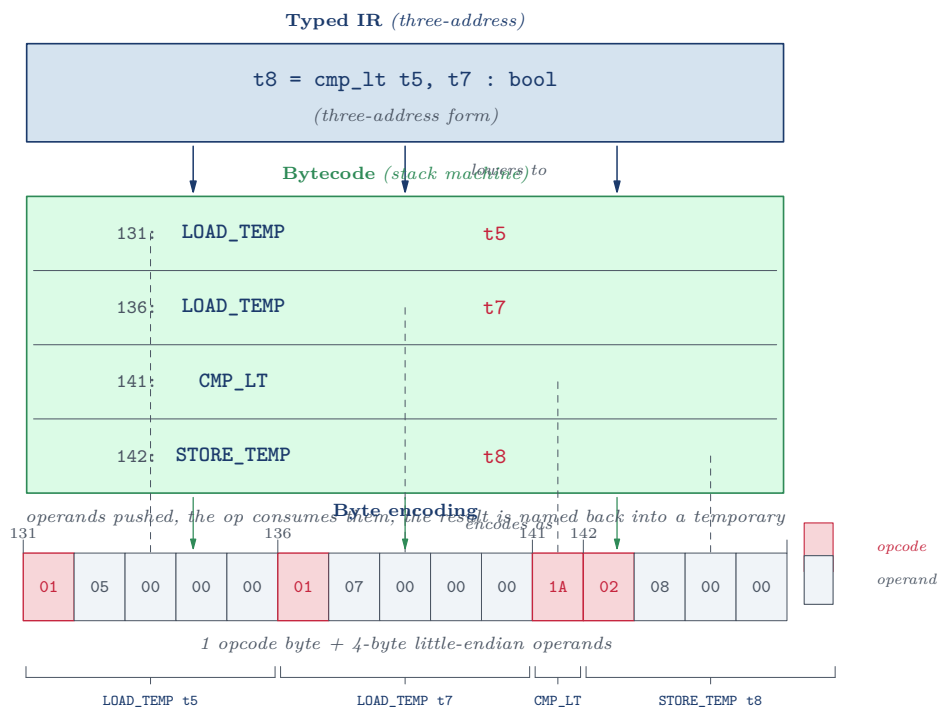


Figure 12.1: The lowering cascade for a single typed-IR instruction. The three-address comparison `t8 = cmp_lt t5, t7` becomes four stack-machine bytecode instructions (middle): two `LOAD_TEMP`s push the operands, `CMP_LT` consumes them and pushes the result, and `STORE_TEMP` names that result back into a temporary. Each instruction encodes (bottom) as one opcode byte — shown in accent — optionally followed by four little-endian operand bytes. The comparison `CMP_LT` takes no operands and so occupies a single byte; `LOAD_TEMP 5` occupies five.

Two design decisions are visible in that figure, and both deserve a word. The first is that we kept the IR's *temporaries*. A purist stack machine would have no `LOAD_TEMP` or `STORE_TEMP` at all: the result of the comparison would simply stay on the stack until the next instruction that needs it, and a *stack-scheduling* pass would arrange every value's lifetime so the stack always held the right thing at the right moment. That is how a hand-written stack-machine compiler squeezes out the loads and stores. We deliberately did not do this. Our IR is three-address and its temporaries are already named, so keeping them — as a per-frame array of integers, a small register file the `LOAD_TEMP` and `STORE_TEMP` opcodes index into — makes the lowering a direct, obviously-correct transcription rather than a scheduling puzzle. The cost is a few extra opcodes per instruction; the benefit is that the bytecode reads as a line-by-line image of the IR, which is exactly what we want in a teaching VM.

Key insight

A three-address instruction becomes postfix on a stack. The translation from three-address code to stack code is the translation from *infix-with-names* to *postfix*. Where the IR writes `t8 = cmp_lt t5, t7` — naming both inputs and the output — the stack code writes “push t5, push t7, compare” and lets position on the stack do the work that names did in the IR. This is the same Reverse Polish Notation a calculator uses, and it is why stack code is so compact: an opcode that always finds its arguments in the same place needs no room to say where they are. The price is that the operands must arrive in the right order, which is precisely what the lowerer guarantees.

The second decision is the encoding itself. Each opcode is one byte — the opcode set has a few dozen entries, comfortably inside a byte's range — and each operand that an opcode needs (a temporary index, a constant, a jump target, an array size) is four bytes, stored little-endian, just as words are stored in the VM's memory. So `CMP_LT`, which takes no operands, is a single byte; `LOAD_TEMP 5` is five. The whole function becomes one flat `byte[]`, and a position in that array — the program counter — is all the VM needs to know where it is.

Bytes, words, and an honest confession. A real production VM packs its bytecode tightly: operands are often a single byte, or variable-length, and a clever encoding keeps the common instructions short to stay friendly to the processor's instruction cache. Our encoding is more relaxed — a fixed four bytes per operand — because clarity matters more than density in a VM you are reading to learn from. Nothing in the design depends on the choice; the disassembler in [Listing 9](#) would look the same with packed operands, and the dispatch loop would change only in how it advances the program counter. We mention it so that the byte counts in this chapter are read for what they are: a faithful measure of *this* VM, not a claim about how small bytecode can be.

With the opcode set settled and the encoding chosen, there is nothing left to design. The remaining question is mechanical: given a block of IR, emit the right run of bytes. That is

the lowerer’s job, and it is where we turn next.

12.3 Lowering the IR to bytecode

Lowering the IR to bytecode is a second code generator. It has the same shape as the FRISC back end of [Chapter 8](#) — walk the IR, emit instructions — but it targets a virtual instruction set instead of a real one, and that makes it markedly simpler. There are no registers to allocate, no spill slots, no instruction selection: a three-address operation maps to a fixed little run of stack opcodes, the same way every time. The lowerer lives in one class, `IrToBytecodeCompiler`, and its core is a recursive walk that, for each right-hand side, emits code to leave that right-hand side’s value on top of the operand stack. [Algorithm 12.1](#) states it.

Algorithm 12.1: Lowering an Rhs to bytecode. Each case emits code that leaves the value on the operand stack; `lowerValue` emits `LOAD_TEMP` for a temporary or `PUSH_CONST` for a constant. An `Assign` wraps this with a trailing `STORE_TEMP` to its destination.

Input: a right-hand side r

Output: bytecode appended that leaves r ’s value on the operand stack

```

1 switch kind of r do
2   case ConstRhs k do
3     | emit(PUSH_CONST, integer encoding of k)
4   case AddrOfSymbol s do
5     | emit(ADDR_SYM, index of s)
6   case Load a :  $\tau$  do
7     | lowerValue(a)
8     | if  $\tau$  is scalar then emit(LOAD_WORD or LOAD_BYTE)
9   case BinOp  $\odot, l, r'$  do
10    | lowerValue(l); lowerValue(r'); emit(opcode for  $\odot$  (integer or Q16.16)
11    | )
12  case AddrIndex b, x, sz do
13    | lowerValue(b); lowerValue(x)
14    | if this index is checked then emit(ADDR_INDEX_CHK, sz, array size)
15    | else emit(ADDR_INDEX, sz)
16  case Call g,  $a_1 \dots a_k$  do
17    | for each  $a_i$  do lowerValue( $a_i$ )
18    | emit(CALL, index of g, k)
19  otherwise do
20    | (CmpOp, UnaryOp, CastOp, AddrField — one opcode each)

```

The structure mirrors the interpreter's `evalRhs` (Algorithm 11.2) almost line for line, which is no accident: both are driven by the same ten-way sealed `Rhs` menu. The difference is what they do at each case. Where the interpreter *computes* a value, the lowerer *emits the opcode that will compute it later*. This is the recurring shift between an interpreter and a compiler, visible in miniature: the same case analysis, but one acts now and the other defers.

Deferring is precisely the point, because it lets the lowerer do, once, all the work the interpreter repeated. Three kinds of static fact get baked into the bytecode here and never reconsidered at run time. Types vanish: the lowerer uses the IR's result types to choose between `LOAD_WORD` and `LOAD_BYTE`, or between integer `MUL` and fixed-point `MUL_Q16`, and then the type is gone — the running VM sees only the chosen opcode. Struct-field offsets and array element sizes become plain operand integers. And the bounds-check decision — which the interpreter recomputes with a small fixed-point dataflow analysis every time it runs a function — is made once, at lowering, and recorded as the choice between two opcodes: `ADDR_INDEX` for an address computation that need not be checked, `ADDR_INDEX_CHK` for one whose result will be loaded from or stored to. The lowerer runs the very same analysis the interpreter does (Section 11.6); it just spends the answer differently.

Key insight

Lowering spends the analysis once. The interpreter and the bytecode lowerer run identical analyses — the same type inference, the same bounds-check fixed point — but the interpreter runs them *every time it executes a function* and the lowerer runs them *once, before execution begins*. The bytecode is where the analysis results are stored. This is the entire economic argument for the extra translation step, reduced to a sentence: compute the facts that do not change once, and write them down.

One piece of the lowerer has no counterpart in the interpreter: control flow must be flattened. The interpreter walks a graph of basic blocks, following an object reference from each block's terminator to its successor. The bytecode has no blocks and no labels — only byte offsets — so a `jmp L1` must become a `JMP` to the *numeric offset* where block `L1`'s code begins. The lowerer handles this in the time-honoured way: a first pass emits every block, recording the offset at which each one starts and leaving each jump's target field blank; a second pass *back-patches* every blank with the now-known offset of its target block. A two-way `br` becomes a single `BR` opcode carrying two offsets, the true target and the false; at run time it pops the condition and sets the program counter to one or the other. Listing 9 shows the result for the opening of the prime sieve — the disassembly the VM can print of its own lowered code.

Read against the IR it came from, the disassembly is almost mechanical: every `addr_of_symbol` is an `ADDR_SYM`, every `load` is a `LOAD_WORD`, every `store` is a push of the address, a push of the value, and a `STORE_WORD`. The block labels survive only because the disassembler keeps a side-table mapping offsets back to the labels they came from; the running VM never sees them. What it sees is a flat array of bytes and a program counter — which is all it needs.

```

3  .func main (params=0, temps=62, code=1114 bytes)
4  L0:
5      0: ADDR_SYM      local:count
6      5: STORE_TEMP    t0
7      10: LOAD_TEMP    t0
8      15: PUSH_CONST   #0
9      20: STORE_WORD
10     21: ADDR_SYM      local:i
11     26: STORE_TEMP    t1
12     31: LOAD_TEMP    t1
13     36: PUSH_CONST   #0
14     41: STORE_WORD
15     42: JMP          L1
16  L1:
17     47: ADDR_SYM      local:i
18     52: STORE_TEMP    t2
19     57: LOAD_TEMP    t2
20     62: LOAD_WORD
21     63: STORE_TEMP    t3
22     68: LOAD_TEMP    t3
23     73: PUSH_CONST   #200
24     78: CMP_LE
25     79: STORE_TEMP    t4
26     84: LOAD_TEMP    t4
27     89: BR           L2, L4

```

Listing 9: The lowered bytecode for the first two blocks of `real_prime_sieve`, as the VM disassembles it. Each line is prefixed with its byte offset. Block `L0` initialises `count` and `i` to zero — notice the `ADDR_SYM`, `STORE_TEMP`, `LOAD_TEMP`, `PUSH_CONST`, `STORE_WORD` pattern that lowers a single store of `0` to a local. Block `L1` loads `i`, pushes `200`, compares with `CMP_LE`, and ends in a `BR` whose two targets, `L2` and `L4`, are the back-patched offsets of the loop body and the loop exit.

12.4 The dispatcher loop

With the bytecode in hand, the machine that runs it is astonishingly small. It holds a program counter, an operand stack, and a stack of call frames, and it loops: fetch the opcode byte at the program counter, advance past it, and switch on it to do the work. There is no `instanceof`, no hash-map probe, no graph to walk — the program counter is an index into a byte array, and the switch is over a small integer. [Algorithm 12.2](#) is the whole engine, with the arithmetic cases collapsed for brevity.

This loop is the counterpart of the interpreter’s block-walker ([Algorithm 11.1](#)), and the contrast between them is the contrast of the whole chapter. The interpreter’s control flow was graph traversal: hold a reference to the current block, run its instructions, follow its terminator to the next block. The VM’s control flow is a single mutable integer: `JMP` assigns the program counter, `BR` assigns it conditionally, and “fall through to the next instruction” is just the implicit `pc + 1` of the fetch. There is no notion of a basic block at run time at all — the blocks dissolved into offsets at lowering, and the loop neither knows nor cares where one block ends and the next begins. [Figure 12.2](#) draws the four pieces of the running machine and how they talk to one another.

The dispatch technique matters enough to name. Ours is a *switch dispatch*: a single `switch` statement on the opcode, which the Java compiler turns into a jump table. It is the simplest technique and the only one Java offers directly. Faster interpreters written in C reach for tricks Java cannot express, and it is worth knowing what they are.

Switch, computed goto, and threaded code. A switch-based dispatcher does one indirect jump per instruction, through the jump table, and then jumps *back* to the top of the loop — two jumps the branch predictor often mispredicts because every opcode funnels through the same site. *Computed-goto* dispatch, a GNU C extension, gives each opcode handler its own copy of the “fetch the next opcode and jump to its handler” epilogue, so the processor sees many distinct jump sites and predicts each far better; the handler addresses live in a table indexed by opcode. *Token-threaded* and *direct-threaded* code push this further, storing in the bytecode not opcodes but the handler addresses themselves, eliminating the table lookup entirely [10]. Ertl and Gregg measured these techniques carefully and found the threaded variants roughly twice as fast as a switch on the processors of their day [26]. None of them is available in portable Java, which has neither `goto` nor first-class labels, so our VM uses the switch and we note the rest as the road not taken. The lesson is that the dispatch loop’s *shape* — one tight, predictable inner loop — is where a bytecode interpreter’s speed is won or lost.

For a machine we are reading to understand rather than profiling for production, the plain switch is exactly right, and the faster techniques belong in a sidebar rather than tangled into the loop’s logic. That logic is now complete but for one thing: it does not yet know

Algorithm 12.2: The dispatcher. Each iteration fetches one opcode and switches on it; `readOperand` reads the next four bytes and advances the program counter. Control flow is a numeric assignment to `pc`, not a graph traversal. The arithmetic and comparison opcodes share one shape: pop two, push the combination.

Input: the current call frame, holding code, program counter `pc`, and temporaries

Output: the value returned by the entry function

```

1 while true do
2    $op \leftarrow code[pc]; pc \leftarrow pc + 1$  (fetch and advance)
3
4   switch  $op$  do
5     case PUSH_CONST do
6        $\_ \leftarrow push(readOperand())$ 
7     case LOAD_TEMP do
8        $\_ \leftarrow push(temps[readOperand()])$ 
9     case STORE_TEMP do
10       $\_ \leftarrow temps[readOperand()] \leftarrow pop()$ 
11    case LOAD_WORD do
12       $\_ \leftarrow push(loadWord(pop()))$ 
13    case STORE_WORD do
14       $v \leftarrow pop(); storeWord(pop(), v)$ 
15    case ADD, CMP_LT, ... do
16       $b \leftarrow pop(); a \leftarrow pop(); push(a \odot b)$ 
17    case ADDR_INDEX_CHK do
18       $sz \leftarrow readOperand(); n \leftarrow readOperand(); x \leftarrow pop(); b \leftarrow pop()$ 
19      if  $x < 0$  or  $x \geq n$  then trap with code  $-6$ 
20       $\_ \leftarrow push(b + x \times sz)$ 
21    case JMP do
22       $pc \leftarrow readOperand()$ 
23    case BR do
24       $t \leftarrow readOperand(); f \leftarrow readOperand(); pc \leftarrow pop() \neq 0 ? t : f$ 
25    case CALL, RET do
26       $\_ \leftarrow$  see Algorithm 12.3

```

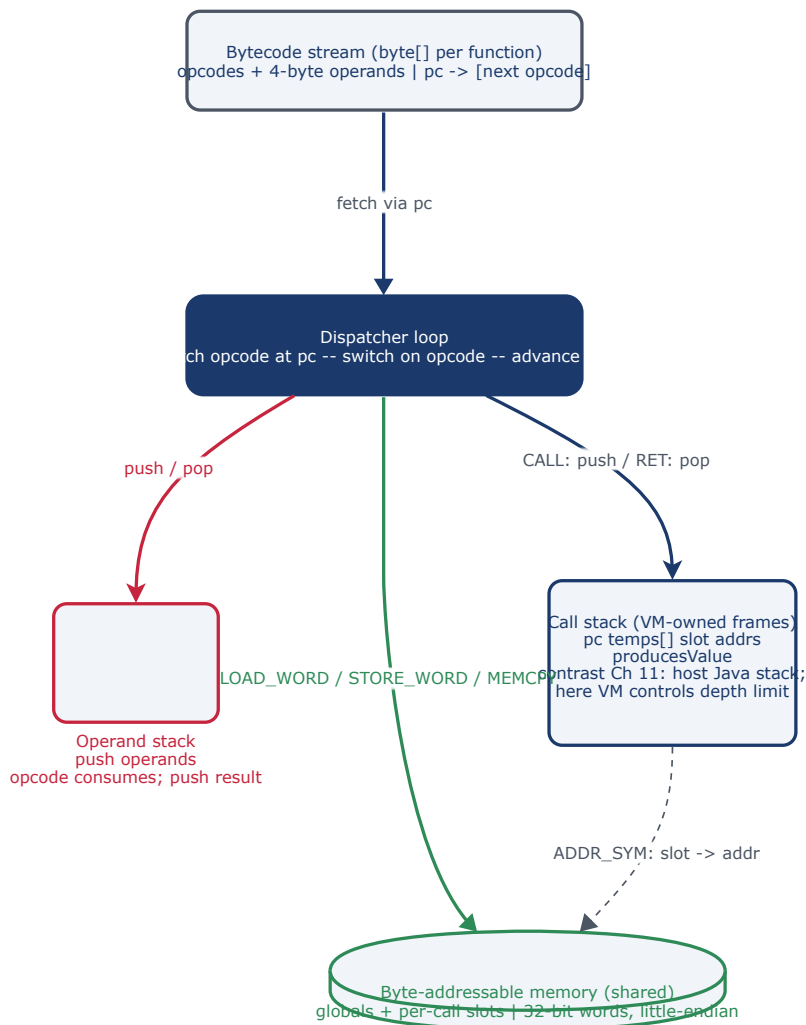


Figure 12.2: The four pieces of the machine and how they talk. The dispatcher loop fetches opcodes from the flat bytecode stream through the program counter, drives all computation through the operand stack, reaches the shared byte-addressable memory for loads, stores, and aggregate copies, and pushes and pops the call stack on CALL and RET. The call stack resolves a symbol's slot name to an address in the same shared memory. Contrast [Chapter 11](#): the interpreter had no call-stack box, because it borrowed the host's.

what to do when a program calls a function. Supplying that is the next section, and it is where the VM stops borrowing and starts building.

12.5 Two stacks of our own

The VM keeps two stacks, and the difference between them is the difference between a value and a function call. The first is the operand stack we have already met: a flat array of integers on which every arithmetic, comparison, address, and memory opcode does its work, growing and shrinking instruction by instruction. The second is the *call stack*: a stack of frames, one per active function call, each holding that call's program counter, its register file of temporaries, and the addresses of its parameter and local slots. This second stack is the one the interpreter did not have, and building it is the heart of the chapter.

When the interpreter met a call, it did the simplest possible thing: it called itself in Java. A recursive IR program became a recursive Java program, and the host runtime maintained the chain of activations on its own stack — we got recursion for free and paid for it with a depth limit we did not control. The VM refuses that bargain. Its dispatch loop never recurses; instead, CALL pushes a new frame onto the call stack and makes it current, and RET pops the finished frame and resumes the one beneath. Algorithm 12.3 gives the protocol.

Algorithm 12.3: How CALL and RET manage the VM's own call stack. A call pushes a frame; a return pops one and leaves the result on the operand stack for the caller's following STORE_TEMP to collect. The dispatch loop never recurses, so the depth of recursion the VM supports is the depth of *this* stack, not the host's.

Input: a CALL to function g with k arguments, on the operand stack

Output: the call stack advanced into g ; later, g 's value returned to the caller

```

1 on CALL:
2 pop  $k$  argument values off the operand stack into  $argv$ 
3  $F \leftarrow \text{newFrame}(g)$            (allocate  $g$ 's slots in memory, fresh temporaries)
4
5 bindParameters( $argv, F$ )           (scalars by value, aggregates by copy)
6
7 record on  $F$  whether the caller wants a value
8 push  $F$  onto the call stack; make it current
9 on RET (value  $v$  on operand stack):
10 pop the finished frame off the call stack
11 if the call stack is now empty then return  $v$            (the entry function returned)
12
13 if the finished frame's caller wanted a value then push( $v$ )
14 resume the caller           (its program counter sits just past the CALL)
15
```

The mechanics repay a careful look, because they are exactly what the host language hid from the interpreter. Arguments are passed on the operand stack: the lowerer emits a `lowerValue` for each argument before the `CALL`, so by the time the opcode runs, the k arguments sit on top of the stack, and `CALL` pops them. Slot allocation and parameter binding are identical to the interpreter's ([Algorithm 11.3](#)), because the VM models the same byte-addressable machine: each call reserves fresh memory for its slots, scalars are written by value, and aggregates are copied byte-for-byte from the address the caller passed. The one genuinely new wrinkle is the return value. When `RET` pops a frame, the caller is waiting at the instruction just past its `CALL`, which the lowerer arranged to be a `STORE_TEMP` for a value-producing call. So `RET` pushes the returned value onto the operand stack, and the caller's next instruction collects it. A void call wants no value, and the frame remembers that, so `RET` discards it instead. [Figure 12.3](#) catches all three structures mid-recursion: two live frames for the same recursive function, the shared operand stack between them, and the memory in which each call's slots are reserved.

Why go to this trouble for something the host gave us free? Because owning the stack means controlling it. A deeply recursive program that would overflow the Java thread's stack under the interpreter — raising a `StackOverflowError` from somewhere deep in `interpretFunction` — runs on the VM until *its* call stack is exhausted, a limit the VM sets and can report cleanly. The two machines genuinely differ here, exactly as the [Section 11.5](#) sidebar foretold: not on the value a program computes, but on how deep a recursion each can sustain and how gracefully it fails when it cannot. The VM trades the interpreter's borrowed convenience for control, which is the trade every serious language runtime eventually makes — a runtime that cannot bound its own stack cannot, for instance, run untrusted code safely.

The operand stack is not the call stack. It is easy to conflate the two, because both are stacks and both grow during a call. They are different machines. The operand stack holds *values* mid-computation — the two integers a `CMP_LT` is about to compare — and it rises and falls many times within a single instruction's worth of IR. The call stack holds *frames* — activation records — and changes only at a `CALL` or a `RET`. A useful picture: the call stack is the table of contents of the running program, one entry per nested call, while the operand stack is the scratch paper the current call is doing its arithmetic on. The JVM draws exactly this distinction, with a per-frame operand stack and a separate frame stack; ours shares one operand-stack array across frames for simplicity, carving each call its own region above the caller's.

Between them, the two stacks give the VM everything it needs to run a program from its first instruction to its last. What they do not yet guarantee is that it runs the program *correctly* — that the value it computes is the value the interpreter and the FRISC compiler compute for the same IR. Earning that agreement is the next section's concern.

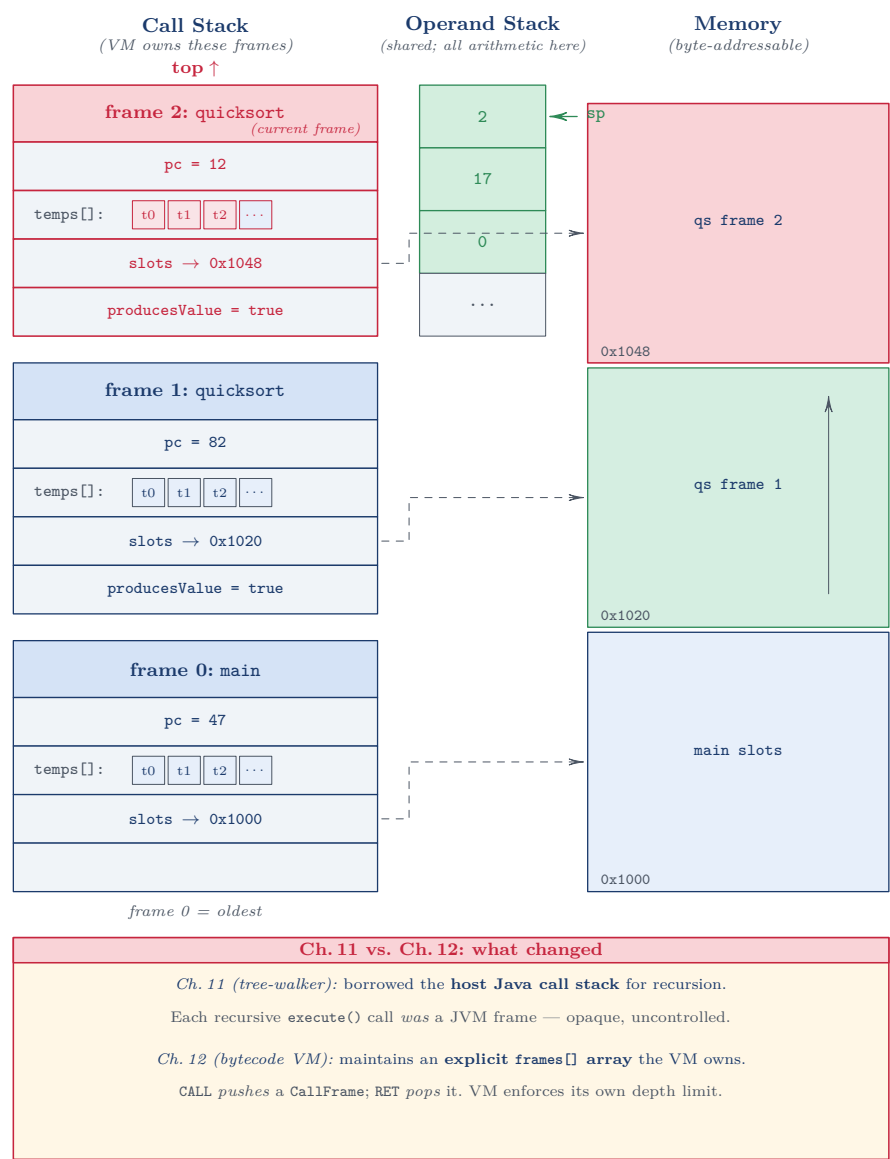


Figure 12.3: Three structures the VM manages itself, captured inside a recursive call. Left: the call stack, three frames deep, with two frames for the *same* recursive function — each with its own program counter, its own register file of temporaries, and its own slot addresses. Centre: the single shared operand stack, where all computation happens. Right: the shared byte-addressable memory, in which each call’s slots are allocated upward. In [Chapter 11](#) the frames on the left were Java stack frames; here they are entries in a stack the VM owns.

12.6 Faithful to the same machine

A second back end is only useful if it agrees with the first. The VM must compute the same values the interpreter does and fail where the interpreter fails, or it is not a model of the IR but a separate language that happens to resemble it. Faithfulness here is not automatic — it is engineered, by making every behavioural choice the same one the interpreter made.

The shared decisions run all the way down. The VM's memory is the same flat, byte-addressable map allocated upward from `0x1000`, with the same little-endian word layout. Its fixed-point arithmetic is the same: `MUL_Q16` computes `(long) a * b >> 16` and `DIV_Q16` computes `((long) a << 16) / b`, bit for bit the routines the interpreter inlined and the FRISC helpers `F_FMUL` and `F_FDIV` implement. Integer division or modulo by zero returns zero rather than trapping, matching the defensive convention everywhere else in the project. An out-of-bounds array access caught by `ADDR_INDEX_CHK` raises the same trap code `-6` that `L_BOUNDS_ERROR` reports on FRISC and that the interpreter returns. And a runaway program trips a watchdog — here a limit on dispatched instructions rather than executed IR steps, scaled up because the bytecode is finer-grained, but the same idea: a development tool must not wedge.

Key insight

Three back ends now triangulate the IR. The interpreter gave us a differential oracle: two readers of the IR that should always agree. The VM makes three — the FRISC code generator, the tree-walking interpreter, and the bytecode VM, sharing nothing but the `IrProgramModel` they each consume. Run a program three ways. If all three agree, our confidence is far higher than any pair could give; if one disagrees with the other two, the outlier is almost certainly the one with the bug. Across the project's full corpus of test programs, the VM's return value matches the interpreter's on every single one — a regression test the build runs automatically, and the strongest evidence we have that the bytecode lowerer and the dispatch loop are correct.

That agreement is worth pausing on, because it was not free and it was not guaranteed. The first versions of any lowerer get the operand order of a non-commutative operation backwards, or check the wrong `addr_index`, or sign-extend a byte the interpreter zero-extended. Every such bug surfaced the same way: a program whose VM result differed from its interpreter result, with the disagreement pointing straight at the opcode that was wrong. The third reader does not merely raise our confidence in the answer; it sharpens every bug into a one-line failure report. This is the practical reason a teaching compiler benefits from more than one back end — they keep each other honest.

12.7 What the extra step buys

We can now weigh the bytecode VM against the interpreter on the same terms we used in [Section 11.7](#): a reproducible count of work done, not wall-clock time. The interpreter's

unit was the *step* — one IR instruction or terminator executed. The VM’s natural unit is the *dispatch* — one bytecode opcode fetched and run. Because one three-address IR instruction lowers to several stack opcodes, the VM necessarily dispatches *more* times than the interpreter steps. Figure 12.4 shows both counts across a spread of programs.

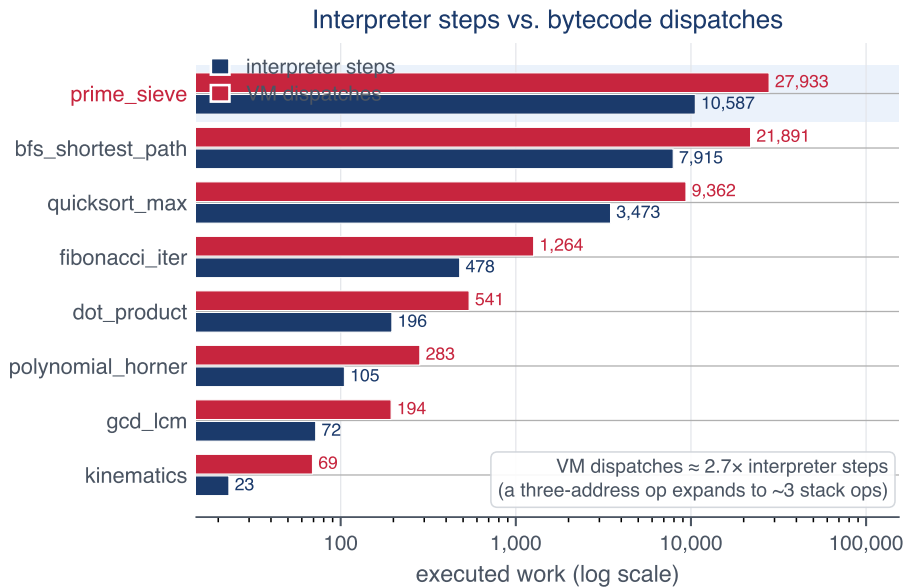


Figure 12.4: For eight programs from the test suite, the interpreter’s executed steps (in blue) against the VM’s dispatched bytecode instructions (in accent), on a logarithmic axis. The VM consistently dispatches about 2.7 times as many instructions, because a single three-address IR operation expands into roughly three stack opcodes — two pushes and an operation, or a push and a store. The anchor `real_prime_sieve` is highlighted: its 10,587 interpreter steps become 27,933 VM dispatches. This is a count of work, not a measure of time.

The ratio is remarkably stable. Across programs as different as a seventy-step `gcd` and a ten-thousand-step sieve, the VM dispatches close to 2.7 times as many instructions as the interpreter executes steps, because the expansion factor is a property of the lowering — how many stack opcodes a three-address instruction becomes — not of the program. At first glance this looks like the VM doing nearly three times the work, which would be a strange thing to call an improvement. The resolution is that the VM’s dispatches are far *cheaper* than the interpreter’s steps. A dispatch is a fetch of one byte from a flat array, a switch on a small integer, and a push or pop on an array-backed stack. An interpreter step is an instance of cascade over a sealed hierarchy, a hash-map lookup for each operand temporary, and a reference chased to the next block. More dispatches, each a fraction of the cost — that is the bytecode bargain.

Key insight

More instructions, each far cheaper. The bytecode VM does not win by doing fewer things; it does more, smaller things. It trades the interpreter’s expensive, structure-examining steps for a larger number of trivial, uniform dispatches over a flat array. Whether that trade pays off in wall-clock time depends on the host, the dispatch technique, and the program — a measurement we make properly in [Chapter 13](#), where all three pipelines run head to head. The claim of *this* chapter is narrower and surer: the bytecode moves the cost of understanding the program out of the inner loop, which is the precondition for any of the speed a real VM delivers.

Optimisation helps the VM for the same reason it helped the interpreter, and the data confirm it. Lowering the optimised prime-sieve IR rather than the unoptimised one drops the dispatch count from 27,933 to 26,965 and shrinks the bytecode from 1,114 bytes to 1,079, because the optimiser removed instructions that the lowerer would otherwise have turned into opcodes. The optimiser neither knows nor cares that a VM will run its output; it removes redundant loads and folds constants because those make *any* execution shorter, and the VM, like the interpreter and the compiled program, simply inherits the saving. Note what we are *not* claiming. We have not measured FRISC cycles against VM dispatches, because they are different units on different machines; the honest three-way comparison of native code, interpreter, and VM is the business of the next chapter. Here we have only established the VM’s own cost structure — more dispatches, each cheap — and shown that it responds to optimisation exactly as the other pipelines do.

12.8 Worked example: the sieve as bytecode

Let us run the back-end anchor, `real_prime_sieve` ([Listing 10](#)), all the way through the VM and watch it work. The program is the sieve of Eratosthenes over the integers up to two hundred: it marks composites in a character array and counts the primes that remain, returning 46. We have carried it since [Chapter 7](#) as the running example of the back end; now we run it without generating any FRISC at all, by lowering its IR to bytecode and dispatching:

```
1 | ./run.sh run-vm book/listings/real_prime_sieve/intermediate_00.ir
```

which prints Return value: 46 and Dispatched instructions: 27933. The right answer, and a precise count of the work. Asking for a trace with `--trace-vm` prints one line per dispatch; [Listing 11](#) shows the opening.

The trace is the dispatch loop made visible, one line per turn. The opening five dispatches (steps 1–5) run the first store of block L0: `ADDR_SYM` pushes the address of `count` (`t0 = 4308`), a `STORE_TEMP` names it, a `LOAD_TEMP` pushes it back, `PUSH_CONST` pushes the initial 0, and `STORE_WORD` writes it to memory — the five-opcode signature of storing zero to a local that


```
1  char isPrime[201];
2
3  int main(void) {
4      int i;
5      int p;
6      int count = 0;
7
8      for (i = 0; i <= 200; i++) {
9          isPrime[i] = 1;
10     }
11     isPrime[0] = 0;
12     isPrime[1] = 0;
13
14     p = 2;
15     while (p * p <= 200) {
16         if (isPrime[p]) {
17             int j = p * p;
18             while (j <= 200) {
19                 isPrime[j] = 0;
20                 j = j + p;
21             }
22         }
23         p++;
24     }
25
26     for (i = 2; i <= 200; i++) {
27         if (isPrime[i]) {
28             count++;
29         }
30     }
31
32     return count;
33 }
```

Listing 10: The back-end anchor: the sieve of Eratosthenes up to two hundred, which returns the prime count 46, carried since [Chapter 7](#) and now run directly from its IR as bytecode.

```

1      Execution trace:
2      [1] main@0 ADDR_SYM 0 | sp=0
3      [2] main@5 STORE_TEMP 0 | sp=1 top=4308
4      [3] main@10 LOAD_TEMP 0 | sp=0
5      [4] main@15 PUSH_CONST 0 | sp=1 top=4308
6      [5] main@20 STORE_WORD | sp=2 top=0
7      [6] main@21 ADDR_SYM 1 | sp=0
8      [7] main@26 STORE_TEMP 1 | sp=1 top=4300
9      [8] main@31 LOAD_TEMP 1 | sp=0
10     [9] main@36 PUSH_CONST 0 | sp=1 top=4300
11     [10] main@41 STORE_WORD | sp=2 top=0
12     [11] main@42 JMP 47 | sp=0
13     [12] main@47 ADDR_SYM 1 | sp=0
14     [13] main@52 STORE_TEMP 2 | sp=1 top=4300
15     [14] main@57 LOAD_TEMP 2 | sp=0
16     [15] main@62 LOAD_WORD | sp=1 top=4300
17     [16] main@63 STORE_TEMP 3 | sp=1 top=0
18     [17] main@68 LOAD_TEMP 3 | sp=0
19     [18] main@73 PUSH_CONST 200 | sp=1 top=0
20     [19] main@78 CMP_LE | sp=2 top=200
21     [20] main@79 STORE_TEMP 4 | sp=1 top=1
22     [21] main@84 LOAD_TEMP 4 | sp=0
23     [22] main@89 BR 622 751 | sp=1 top=1
24     [23] main@622 ADDR_SYM 3 | sp=0
25     [24] main@627 STORE_TEMP 5 | sp=1 top=4096
26     [25] main@632 ADDR_SYM 1 | sp=0
27     [26] main@637 STORE_TEMP 6 | sp=1 top=4300
28     [27] main@642 LOAD_TEMP 6 | sp=0
29     [28] main@647 LOAD_WORD | sp=1 top=4300
30     [29] main@648 STORE_TEMP 7 | sp=1 top=0
31     [30] main@653 LOAD_TEMP 5 | sp=0
32     [31] main@658 LOAD_TEMP 7 | sp=1 top=4096
33     [32] main@663 ADDR_INDEX_CHK 1 201 | sp=2 top=0

```

Listing 11: The first dispatches of the sieve on the bytecode VM. Each line shows the dispatch count, the function and byte offset, the opcode and its operands, and the operand stack's depth and top entering the instruction. Steps 1–11 run block L0, initialising count and i. Steps 12–22 run the loop test L1. At step 32, ADDR_INDEX_CHK computes a checked address into the sieve array — element size 1, array size 201 — the bounds check the lowerer decided this access needed.

we saw in the disassembly. Steps 6–10 repeat the pattern for `i`, and the `JMP` at step 11 jumps to the loop test. There, `i` and the constant 200 are pushed, `CMP_LE` compares them and pushes 1, and the `BR` at step 22 takes the true edge into the loop body. Watch the operand stack column rise and fall as it goes: it holds at most a couple of values at a time, exactly the scratch paper the sidebar described.

The detail worth lingering on is step 32. The IR computes an address into the sieve array with an `addr_index`, and because that address feeds a `store`, the lowerer marked it for checking and emitted `ADDR_INDEX_CHK` with two operands: the element size, 1, since the array holds chars, and the array size, 201. At run time the opcode pops the index and the base, verifies the index lies in $[0, 201)$, and pushes the computed address; an index out of range would trap with code -6 and become the program's result, just as it would on FRISC. The interpreter made the very same decision, but it remade it every time it entered the function; the VM made it once, at lowering, and wrote it into the opcode. That single difference — decide once and record, versus decide every time — is the chapter in miniature.

From there the VM simply keeps dispatching. It rounds the inner and outer loops of the sieve, marking composites through checked stores and counting primes, following `BR` and `JMP` by assigning its program counter, until the outer loop's condition finally fails and the final `RET` pops the last frame off the call stack and hands back the value on the operand stack. No registers were allocated, no FRISC was emitted, no `a.out` was written. We took the same typed IR the compiler would have lowered to assembly, lowered it instead to a flat bytecode, and ran it on a machine of our own — and in 27,933 dispatches it told us that there are 46 primes below two hundred, the very answer the compiled program and the interpreter both produce. Three back ends, one IR, one result.

Practice

Exercise 12.1 (Applied · ★★★)

Counting the expansion. Using the disassembly in [Listing 9](#), count the bytecode opcodes that block L1 lowers to, and compare with the number of IR instructions and the one terminator in the corresponding IR block. What is the expansion factor for this block, and how does it compare with the suite-wide average of about 2.7 reported in [Figure 12.4](#)? Which single IR instruction shape expands into the most opcodes, and why?

Exercise 12.2 (Applied · ★★★)

Reading the byte encoding. [Figure 12.1](#) encodes `LOAD_TEMP 5` as the five bytes 01 05 00 00 00. Using the same scheme — one opcode byte, four little-endian operand bytes per operand — write out the byte encoding of the instruction `STORE_TEMP 8` (its opcode byte is 02) and of `ADDR_INDEX_CHK` with element size 1 and array size 201 (its opcode byte is 06). How many bytes does each occupy? Why does the little-endian

order put the least-significant byte first?

Exercise 12.3 (Applied · ★★★)

Why two stacks. Trace the operand-stack column in [Listing 11](#) from step 17 to step 22 and confirm it returns to where it began. Then explain, in your own words, why the VM cannot use a single stack for both operands and call frames. What would go wrong if a CALL happened while intermediate operand values were still on the stack and the two stacks were one?

Exercise 12.4 (Applied · ★★★)

Where the back ends may differ. [Section 12.5](#) claims the interpreter and the VM agree on every value but may differ on how deep a recursion each sustains. Explain the mechanism: which stack does the interpreter exhaust on deep recursion, which does the VM exhaust, and why are their limits different? Then describe how you would measure each limit empirically using a recursive program, and predict which back end would survive a deeper recursion on a typical JVM.

Exercise 12.5 (Applied · ★★★)

The cost of a checked access. The lowerer emits ADDR_INDEX_CHK only for address computations whose result is loaded from or stored to, and plain ADDR_INDEX otherwise — the decision of [Section 11.6](#), made once at lowering. Suppose a program indexes the same array a thousand times in a hot loop. How many times does the VM perform the bounds comparison, and how many times did it perform the *analysis* that decided to check? Contrast this with the interpreter. What does this say about where the two designs spend their effort?

Project

Exercise 12.6 (Lab · ★★★)

Project: a bytecode VM for tinyC. Build a stack-based bytecode compiler and virtual machine for the tinyC IR you produced in [Chapter 6](#)’s project, so that the same IR can now be run three ways — compiled, tree-walked, and on a bytecode VM — and all three cross-checked against one another.

Open the `ch12-vm/starter/` module in the companion repository, under the `com/frisccc/tinyc/vm/` package. The opcode set (`Opcode`), the encoded-program model (`Bytecode`), and the byte-addressable `Memory` are provided; tinyC is int-only, so there is no fixed-point arithmetic, no aggregate copying, and no bounds-check analysis to write, and your opcode set is correspondingly small.

Three // TODO: implement markers await you, one per algorithm in this chapter:

TODO 1 `lowerRhs` and `lowerInstruction`: the lowering of [Algorithm 12.1](#), restricted to tinyC’s shapes — constants, `addr_of_symbol`, `load`, the integer `BinOps` and `CmpOps`, `UnaryOp`, and `Call` — each leaving its value on the operand stack, with a trailing `STORE_TEMP` for an `Assign`.

TODO 2 block flattening: emit each block, record its start offset, and back-patch every `JMP` and `BR` target to the numeric offset of its destination block.

TODO 3 the dispatch loop of [Algorithm 12.2](#) and the call protocol of [Algorithm 12.3](#): fetch, switch, and push or pop the explicit operand and call stacks — *without* recursing in the host language, so that your VM, not the JVM, owns the call depth.

The provided tests run a battery of tinyC programs through your VM and through both your interpreter (from [Chapter 11](#)’s project) and the reference compiled path, and assert that all three return values *match* — the three-way triangulation of [Section 12.6](#) in miniature. As a stretch goal, add a `--disassemble` flag that prints your bytecode in the style of [Listing 9](#), and a `--trace` flag in the style of [Listing 11](#); then write a deliberately deep recursion and find the depth at which your VM’s call stack overflows, confirming it differs from where your tree-walker’s host stack would.

Your turn

Open `frisccc-companion/ch12-vm/starter/` and build the bytecode compiler and virtual machine the Project describes. The `Opcode` enum, the `Bytecode` model, and the byte-addressable `Memory` are provided; the three things left to you are the `lowerRhs` lowerer of [Algorithm 12.1](#), the block-flattener that back-patches jump offsets, and the dispatch loop of [Algorithm 12.2](#) with the call protocol of [Algorithm 12.3](#) — a loop that owns its own call stack rather than recursing in the host. When the provided tests pass, the same tinyC IR you have carried since [Chapter 6](#) will run three ways and agree with itself every time. The reference build waits in `frisccc-companion/ch12-vm/solution/`.

Notes and further reading

The definitive modern tour of bytecode virtual machines, and the companion to this book’s whole interpreter–VM arc, is the second half of Nystrom’s *Crafting Interpreters* [67], which builds a single-pass bytecode compiler and a stack-based VM for the same language its first half tree-walks — precisely the two chapters this part of the book mirrors. Smith and Nair’s *Virtual Machines* [76] is the broad systems-level treatment, situating language VMs like ours among the wider family of virtualisation techniques and drawing the operand-stack / frame-stack distinction we leaned on in [Section 12.5](#).

The performance of the dispatch loop is a small literature of its own. Bell’s one-page note introduced threaded code in 1973 [10]; Ertl and Gregg’s careful measurements of switch, threaded, and replicated dispatch [26] are the reference for why a portable switch leaves speed on the table and what a C implementation would do instead. The stack-versus-register question — whether a VM should pass operands on a stack, as ours and the JVM do, or address them directly, as the Lua and Dalvik VMs do — was put to rest empirically by Shi and colleagues, who found register bytecode executes fewer, if larger, instructions [74]; Chapter 15 takes up what rebuilding our VM that way would change. The recurring theme across all of this work is the one Wheeler’s epigraph names: the bytecode is a level of indirection, and the engineering is all in making that indirection cheap to cross.

The next chapter stops building machinery and starts using it. We have three ways to run a program now — native FRISC, the tree-walking interpreter, and this bytecode VM — and Chapter 13 runs a curated portfolio of programs through all three, where the abstract trade-offs of these two chapters finally become concrete numbers we can lay side by side.

Part V

Putting It Together

Chapter 13

Case studies

“Talk is cheap. Show me the code.”

— Linus Torvalds

By the close of [Chapter 12](#) we had built three ways to run a program. The FRISC code generator of [Chapter 8](#) compiles the typed IR all the way down to machine instructions a real processor could execute. The tree-walking interpreter of [Chapter 11](#) reads the same IR and does what it says, instruction by instruction, computing the answer without emitting any machine code at all. The bytecode VM of [Chapter 12](#) sits between them, lowering the IR to a flat stream of stack opcodes and dispatching through it on a machine of its own design. Three readers of one intermediate representation, sharing nothing but the `IrProgramModel` they each consume. We proved they agree on answers; we have not yet watched them disagree on cost. That is this chapter’s whole business, and the closing words of [Chapter 12](#) promised it: “the abstract trade-offs of these two chapters finally become concrete numbers we can lay side by side.”

We will lay them side by side on five small programs, chosen not because they are large but because each one stresses a different seam in the machinery. A naive recursive Fibonacci spends everything on the call stack and nothing else, so it shows what a function call *costs* when you make twenty-two thousand of them. A Euclidean GCD leans on integer modulo, division, and multiplication — three operators the FRISC has no hardware for — so it shows what happens when a single IR instruction expands into a software loop. A knapsack solver iterates over an array with computed indices and no arithmetic helpers at all, so it isolates the irreducible cost of iteration and bounds-checking. A fixed-point Horner evaluation runs a multiply-accumulate loop in `Q16.16`, and a straight-line kinematics formula does the same arithmetic with no loop at all — and then, at `-01`, vanishes entirely. Between them the five programs cover recursion, helper-bound arithmetic, pure iteration, and the fixed-point runtime, which is most of what the compiler’s cost structure has to teach.

A word first about what we are measuring, because it is easy to measure the wrong thing and draw a confident wrong conclusion from it.

Three currencies, and what they do not buy. Each back end has a natural unit of work, and the three units are not the same. The native pipeline’s unit is the *executed FRISC instruction* — one fetch-decode-execute cycle of the simulator, counted

by stepping it one instruction at a time exactly as the runner in [Chapter 10](#) does. The interpreter’s unit is the *step* — one IR instruction or terminator carried out, the same unit we used in [Chapter 11](#). The VM’s unit is the *dispatch* — one bytecode opcode fetched and run, the unit of [Chapter 12](#). All three count primitive operations, and so all three are honest measures of how much work a back end does to run a program. What they are *not* is interconvertible into wall-clock time. A FRISC instruction runs on a JavaScript simulator; an interpreter step runs on the JVM; a VM dispatch runs on the JVM too but does far less per dispatch than a step does. Counting them tells us about the *shape* of each back end’s cost — where the work goes, and why one program costs a thousand times another — which is a deeper and more durable thing than a millisecond reading on one laptop. Wall-clock timing of the compiler itself is the business of [Chapter 14](#); here we anatomise cost, not speed.

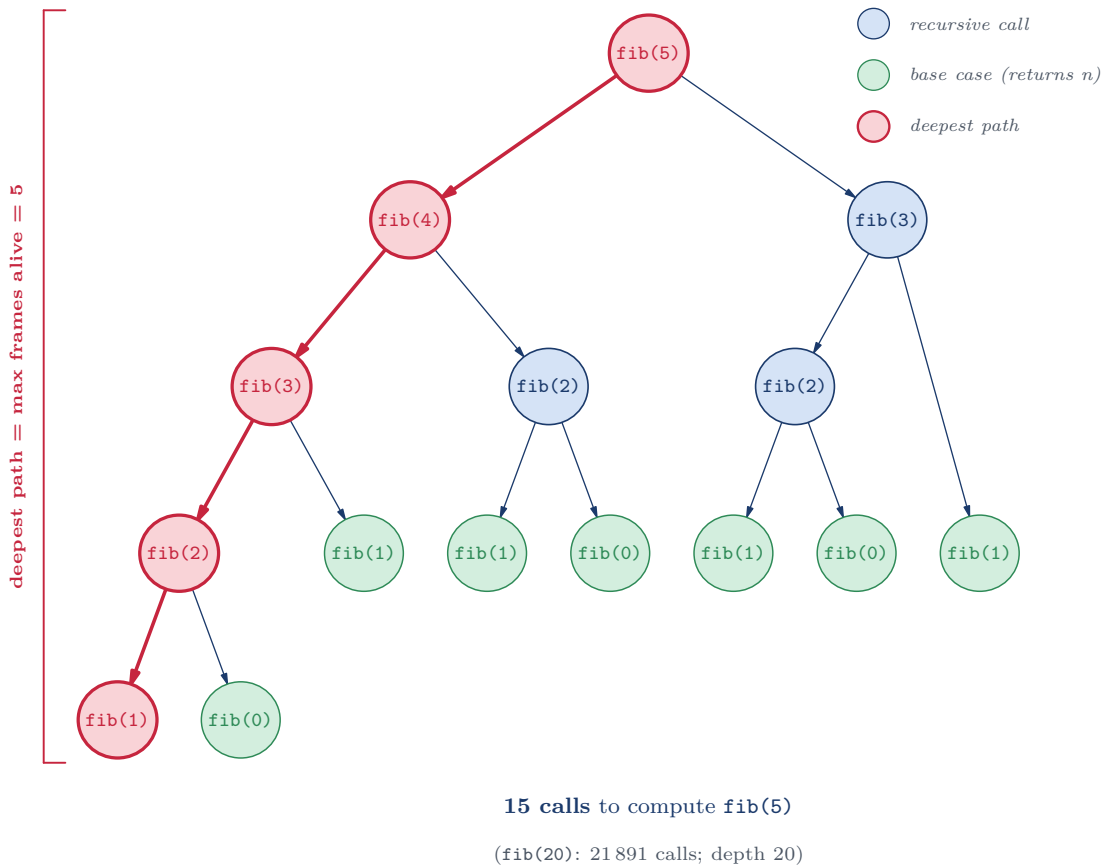
One fact frames everything that follows, and we state it once so we need not repeat it five times: across all five programs, at both -00 and -01, the three back ends returned identical values — thirty runs, one answer per program. The differential oracle of [Chapter 11](#) and [Chapter 12](#) held without exception. So when the numbers below differ by three orders of magnitude between back ends, none of that difference is disagreement about *what* the program computes. It is all about how much work each machine does to compute the same thing. Let us watch.

13.1 Recursive Fibonacci: where the call stack lives

The iterative Fibonacci of [Chapter 3](#) — the front-end anchor we carried through five chapters — walked a loop twenty times and returned 6765. Here is the same number computed the textbook-naïve way, by the recurrence itself ([Listing 12](#)). It returns 6765 as well; what changes is everything underneath.

The IR makes the road visible. The recursive block lowers to two call instructions whose results are added and returned ([Listing 13](#)). There is nothing clever here — no memoisation, no accumulator — and that is the point. Every call to `fib` with $n \geq 2$ spawns two more, so the number of calls grows like the Fibonacci numbers themselves. Computing `fib(20)` this way makes 21,891 calls to `fib` — we counted them, by stepping the simulator and watching the program counter cross the function’s entry — to do work the loop did in twenty iterations.

[Figure 13.1](#) draws the call tree for the small instance `fib(5)`, where the shape is still legible. Two things are worth seeing at once. The tree is *broad*: fifteen nodes to compute `fib(5)`, most of them recomputing values their cousins already found. And the tree is, by comparison, *shallow*: the longest path from root to leaf is only five deep. Breadth is the total number of calls — the work; depth is the maximum number of calls alive at one instant — the call stack. For `fib(20)` the breadth is 21,891 and the depth is 20. The interpreter and the VM differ on exactly one of these two quantities, and it is not the one you might expect.



*Ch. 11 interpreter: frames live on the host Java stack (opaque, depth-limited by the JVM).
 Ch. 12 bytecode VM: frames live in the VM's own `frames[]` array — `CALL/RET` push/pop them explicitly.*

Figure 13.1: The call tree of `fib(5)`: fifteen calls, depth five. Internal nodes recurse; the green leaves are base cases that return immediately. The accent-coloured path is one root-to-leaf chain — the deepest set of frames alive at any single moment, and therefore the call stack's depth. Breadth (total calls) is the work; depth (longest path) is the stack. For the chapter's actual instance `fib(20)`, breadth is 21,891 and depth is 20.

```

1  // EXPECT: 6765
2
3  int fib(int n) {
4      if (n < 2) {
5          return n;
6      }
7      return fib(n - 1) + fib(n - 2);
8  }
9
10 int main(void) {
11     return fib(20);
12 }

```

Listing 12: Naive recursive Fibonacci. The recurrence $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$ is transcribed directly, base case and all. It returns the same 6765 the iterative anchor did, but reaches it by a very different road.

Run the three back ends and the breadth dominates the numbers utterly. The native pipeline executes 1,368,194 FRISC instructions; the interpreter takes 218,909 steps; the VM dispatches 623,888 opcodes. The ratios are the story. Each interpreter step — one IR operation — becomes about 6.25 executed FRISC instructions, and that factor is not arithmetic but *protocol*: every one of the 21,891 calls runs a prologue that saves the frame pointer and reserves slots, a CALL that pushes a return address, an epilogue that tears the frame down, and a RET that pops back. We measured this in [Chapter 8](#); here we see its price multiplied by twenty-two thousand. The native code spends essentially all of its 1.37 million instructions inside F_FIB itself — no arithmetic helpers are called, because addition and subtraction are single hardware instructions — so this program’s cost is call overhead, pure and unalloyed.

Now the quantity the back ends genuinely differ on. The interpreter met each call by calling *itself* in Java, so the chain of twenty live fib activations became twenty nested Java stack frames; the recursion rode the host’s call stack, as [Section 12.5](#) discussed. The VM refused that bargain: its dispatch loop never recurses, and CALL pushes a frame onto an explicit stack the VM owns ([Listing 14](#)). At depth 20 neither machine is troubled — twenty frames is nothing. But the *breadth*, the 21,891 calls, is identical work for both: the interpreter takes one step per IR call and the VM dispatches one CALL opcode, regardless of who owns the stack. Depth is where they differ in *capability*; breadth is where they spend their *time*, and breadth is the same for both.

Key insight

Optimisation is not magic; it cannot unredundant recursion. The optimiser leaves this program completely unchanged: the -00 and -01 IR are byte-for-byte identical, so all three back ends post the same counts at both levels. Constant folding

```
1 .func fib(n:int32):int32
2   .frame locals=0 bytes align=4
3   .slots
4     param n@0:int32
5   .blocks
6   L0:
7     t0 = addr_of_symbol param:n
8     t1 = load t0 : int32
9     t2 = cmp_lt t1, #2:int32 : bool
10    br t2, L1, L2
11  L1:
12    t3 = addr_of_symbol param:n
13    t4 = load t3 : int32
14    ret t4
15  L2:
16    t5 = addr_of_symbol param:n
17    t6 = load t5 : int32
18    t7 = sub t6, #1:int32 : int32
19
20    t8 = call func:fib(t7) : int32
21    t9 = load t5 : int32
22    t10 = sub t9, #2:int32 : int32
23    t11 = call func:fib(t10) : int32
24    t12 = add t8, t11 : int32
25    ret t12
26 .endfunc
```

Listing 13: The IR for fib. Block L0 tests the base case and branches; block L2 is the recursion, where `t8 = call func:fib(t7)` and `t11 = call func:fib(t10)` compute the two sub-results that `t12 = add t8, t11` combines. Two calls per non-base invocation is the entire reason the call count explodes.

```

1  Execution trace:
2  [1] main@0 PUSH_CONST 20 | sp=0
3  [2] main@5 CALL 0 1 | sp=1 top=20
4  [3] fib@0 ADDR_SYM 0 | sp=0
5  [4] fib@5 STORE_TEMP 0 | sp=1 top=4096
6  [5] fib@10 LOAD_TEMP 0 | sp=0
7  [6] fib@15 LOAD_WORD | sp=1 top=4096
8  [7] fib@16 STORE_TEMP 1 | sp=1 top=20
9  [8] fib@21 LOAD_TEMP 1 | sp=0
10 [9] fib@26 PUSH_CONST 2 | sp=1 top=20
11 [10] fib@31 CMP_LT | sp=2 top=2
12 [11] fib@32 STORE_TEMP 2 | sp=1 top=0
13 [12] fib@37 LOAD_TEMP 2 | sp=0
14 [13] fib@42 BR 51 78 | sp=1 top=0
15 [14] fib@78 ADDR_SYM 0 | sp=0
16 [15] fib@83 STORE_TEMP 5 | sp=1 top=4096
17 [16] fib@88 LOAD_TEMP 5 | sp=0
18 [17] fib@93 LOAD_WORD | sp=1 top=4096

```

Listing 14: The opening of the VM trace. At dispatch 2, `main` executes `CALL 0 1` — call function 0 (`fib`) with one argument — and dispatch 3 is already inside `fib` at offset 0. The `BR 51 78` at dispatch 13 takes the false edge to offset 78, the recursive block, because $n = 20$ is not less than 2. Each such `CALL` pushes a frame onto the VM’s own call stack rather than recursing in the host.

finds no constants to fold, copy propagation no copies, and the tiny-function inliner declines to inline a function into itself — inlining a recursive call would not terminate. The redundancy here is *algorithmic*: the program recomputes `fib(18)` thousands of times, and no local, semantics-preserving rewrite of the kind [Chapter 7](#) performs can see that, because seeing it would require memoising results across calls, which changes the program’s memory behaviour. The lesson is bracing and worth keeping: the optimiser shrinks *how* you compute, never *what* you chose to compute. A bad algorithm compiles to fast machine code for a bad algorithm.

Fibonacci, then, is the program where all the cost is the call itself and the optimiser is a bystander. Its mirror image is a program where a single innocent-looking operator hides a loop the optimiser also cannot touch — not because the operator is redundant, but because the hardware cannot perform it. That is the next program.

13.2 GCD and LCM: a helper-dominated cycle budget

We throw Euclid’s algorithm at the compiler next, and at a glance it ought to be the cheapest program in the chapter: three lines of arithmetic ([Listing 15](#)) that reduce $x \bmod y$ until the remainder is zero, wrapped in a least common multiple computed as $(a/g) \cdot b$. Modulo, division, multiplication — the source reads like nothing at all, and the interpreter agrees, running the whole thing in 72 steps to return 420. Then we run the native pipeline,

and it takes 1,457 FRISC instructions to return the same 420 — a twentyfold gap, on a program with barely a loop in it beyond the handful of Euclid iterations. Where did twenty native instructions per IR step come from?

```

1  // EXPECT: 420
2
3  int a = 84;
4  int b = 30;
5
6  int gcd(int x, int y) {
7      int t;
8      while (y != 0) {
9          t = x % y;
10         x = y;
11         y = t;
12     }
13     return x;
14 }
15
16 int main(void) {
17     int g = gcd(a, b);
18     int l = (a / g) * b;
19     return l;
20 }
```

Listing 15: Euclid’s algorithm and a least-common-multiple wrapper. The three operators %, /, and * look free in the source; on a processor without hardware multiply or divide they are anything but.

The IR is as innocent as the source: the loop body is a single `t7 = mod t4, t6` (Listing 16), one three-address instruction the interpreter executes as one step — because on the JVM, % is one Java operator and the host’s processor has a divide instruction. The interpreter inherits the host’s hardware and pays a single step. The FRISC has no such instruction. As Chapter 9 explained, integer multiply, divide, and modulo on the FRISC are *software*: the code generator emits a `CALL` to a runtime helper — `F_MOD`, `F_DIV`, `F_MUL` — and the helper does the arithmetic the long way.

Listing 17 shows what `CALL F_MOD` reaches. The helper is a shift-and-subtract long division: it initialises a bit counter to 32 and loops once per bit, shifting the dividend left, comparing against the divisor, and subtracting when it fits. Thirty-two iterations of several instructions each, plus sign handling, for a single %. The interpreter never sees any of this — it asked the host for a remainder and got one. The native code lives the whole loop, every time.

When we profile the 1,457 executed instructions by routine, the matter settles. The arithmetic helpers account for the overwhelming majority: `F_MOD` alone is 51.9% of all instructions executed, `F_DIV` another 19.9%, and `F_MUL` 4.3% — together 76% of the program’s entire native cycle budget is spent inside three helper routines emulating operators the

```

1  .func gcd(x:int32, y:int32):int32
2  .frame locals=4 bytes align=4
3  .slots
4      local t@0:int32
5      param x@0:int32
6      param y@4:int32
7  .blocks
8  L0:
9      jmp L1
10 L1:
11     t0 = addr_of_symbol param:y
12     t1 = load t0 : int32
13     t2 = cmp_ne t1, #0:int32 : bool
14     br t2, L2, L3
15 L2:
16     t3 = addr_of_symbol param:x
17     t4 = load t3 : int32
18     t5 = addr_of_symbol param:y
19     t6 = load t5 : int32
20     t7 = mod t4, t6 : int32
21
22     t8 = addr_of_symbol local:t
23     store t8, t7 : int32
24
25     t9 = load t5 : int32
26     store t3, t9 : int32
27     t10 = load t8 : int32
28     store t5, t10 : int32
29     jmp L1
30 L3:
31     t11 = addr_of_symbol param:x
32     t12 = load t11 : int32
33     ret t12
34 .endfunc

```

Listing 16: The IR for gcd. The entire loop body reduces to one mod ($t7 = \text{mod } t4, t6$) plus the load/store traffic that moves x, y, and t around. One mod instruction; one interpreter step; one CALL F_MOD on the FRISC.


```

1  L_MOD_B_POS_21
2      MOVE 0, R2          ; remainder
3      MOVE 20, R4         ; bit count (32)
4  L_MOD_LOOP_22
5      SHL R0, 1, R0
6      JP_NC L_MOD_NO_CARRY_23
7      SHL R2, 1, R2
8      OR R2, 1, R2
9      JP L_MOD_AFTER_CARRY_24
10 L_MOD_NO_CARRY_23
11     SHL R2, 1, R2
12 L_MOD_AFTER_CARRY_24
13     CMP R2, R1
14     JP_SLT L_MOD_SKIP_SUB_25
15     SUB R2, R1, R2
16 L_MOD_SKIP_SUB_25
17     SUB R4, 1, R4
18     JP_NE L_MOD_LOOP_22
19     CMP R6, 0
20     JP_EQ L_MOD_SIGN_DONE_26

```

Listing 17: The shift-and-subtract core of `F_MOD`, the software-modulo runtime routine. `MOVE 20, R4` sets the bit counter to 32 (hexadecimal 20); the loop at `L_MOD_LOOP_22` runs once per bit, shifting the dividend left, comparing against the divisor, and conditionally subtracting, until the counter reaches zero at `JP_NE L_MOD_LOOP_22`. One IR `mod` becomes this entire loop. `F_DIV` and `F_MUL` have the same shape.

ISA lacks. The actual logic of Euclid’s algorithm, `F_GCD`, is 17%; `main`, which orchestrates the whole thing, is under 7%. The program does almost no work of its own. It spends three-quarters of its life inside the runtime, performing in software what a desktop processor does in a single instruction.

Key insight

An interpreter step and a native instruction measure different worlds. The interpreter’s step count is a faithful measure of the *algorithm*’s essential operations: 72 steps is genuinely how many IR instructions Euclid executes. But it is a terrible predictor of native cost here, understating it twentyfold, because each step’s true price depends on what the *target* can do in hardware. On a host with `divide`, `mod` is free; on the FRISC it is a 32-iteration loop. The step count tells you the shape of the computation; only knowing the target’s instruction set tells you the bill. This is exactly why a compiler writer cannot reason about performance from the IR alone — the IR is target-independent, and the costliest facts are target-specific. The same gap opens up whenever a language runs on a machine richer than its target: a Python loop that hands its inner arithmetic to a compiled numerical kernel finishes in microseconds, while the identical loop in pure Python takes milliseconds, and the operation count looks much the same either way. The host’s hardware is quietly doing work the counted operations never reveal.

Optimisation helps a little and cannot help much. At -O1 the interpreter’s steps fall from 72 to 57 and the native count from 1,457 to 1,404, as the optimiser removes a few redundant loads around the loop. But the helper share is, if anything, reinforced — it climbs to about 79% as the surrounding integer code is trimmed while the helper calls remain — because the optimiser cannot remove a `mod` that the program genuinely needs, and it has no power to turn a software-emulated operator into a hardware one. The only thing that would move this number is a different *target*: a FRISC with a hardware multiplier and divider would erase three-quarters of this program’s cost at a stroke, a point [Chapter 15](#) returns to. For now the lesson is that some costs live in the instruction set, not the program, and no amount of source-level cleverness reaches them.

Both programs so far have spent their budget on something other than the algorithm — calls in one case, helpers in the other. What does a program look like when the cost *is* the algorithm, when there are no helpers to call and no recursion to explode? That is the next one.

13.3 Knapsack: iteration and the price of a bounds check

The 0/1 knapsack solver in [Listing 18](#) is the cleanest program in the portfolio, in the specific sense that its cost is its algorithm and nothing else. It fills a one-dimensional dynamic-programming table with nested loops, using only addition and comparison — no multiply, no divide, no modulo, so not a single arithmetic helper is called. And every function is inlined into `main` by the time the code generator runs, so there are no `CALLS` either. What is left, when you remove helpers and calls, is the irreducible cost of iteration: loop counters, array indexing, and the bounds checks that guard every array access.

The interesting accesses are the computed ones. Reading `dp[cap - w]` indexes the table at a position the compiler cannot know in advance — it is a subtraction of two run-time values — so the access must be bounds-checked. [Listing 19](#) shows the IR: `t37 = sub t34, t36` computes `cap - w`, and `t38 = addr_index t32, t37, 4` forms the address of the table element at that index, scaled by the four-byte element size. Whether that `addr_index` is checked at run time is decided once, at lowering, by the analysis of [Section 11.6](#) — and because this address feeds a load, it is checked.

The bytecode makes the check concrete. [Listing 20](#) is the lowered form of that same access: two `LOAD_TEMP`s and a `SUB` compute the index, and then the base `LOAD_TEMP t32` and index `LOAD_TEMP t37` are pushed and a single `ADDR_INDEX_CHK` opcode — carrying the element size 4 and the array size 11 as its operands — consumes both, forming the address and verifying the index lies in `[0, 11)` in one opcode. The lowerer decided once that this access needs checking and emitted the checked opcode; the VM performs the comparison at run time, but never re-decides whether to. This is the “decide once, record” economy of [Chapter 12](#) in miniature, and across the whole run the VM executes seven distinct `ADDR_INDEX_CHK` sites.

When we run the three back ends, the numbers come out at the lowest ratio of any program here. The native pipeline executes 6,643 instructions, the interpreter takes 1,563 steps, the

```
1 // EXPECT: 15
2
3 int weights[5] = {2, 3, 4, 5, 9};
4 int values[5] = {3, 4, 5, 8, 10};
5 int dp[11];
6
7 int main(void) {
8     int i;
9     int cap;
10
11     for (cap = 0; cap <= 10; cap++) {
12         dp[cap] = 0;
13     }
14
15     for (i = 0; i < 5; i++) {
16         int w = weights[i];
17         int v = values[i];
18         for (cap = 10; cap >= w; cap--) {
19             int candidate = dp[cap - w] + v;
20             if (candidate > dp[cap]) {
21                 dp[cap] = candidate;
22             }
23         }
24     }
25
26     return dp[10];
27 }
```

Listing 18: A 0/1 knapsack dynamic program. The table `dp` is updated in place from high capacity down to low; the read `dp[cap - w]` uses a *computed* index, which is what makes the bounds-check question interesting.

```

1  L10:
2      t32 = addr_of_symbol global:dp
3      t33 = addr_of_symbol local:cap
4      t34 = load t33 : int32
5      t35 = addr_of_symbol local:w
6      t36 = load t35 : int32
7      t37 = sub t34, t36 : int32
8      t38 = addr_index t32, t37, 4
9      t39 = load t38 : int32
10     t40 = addr_of_symbol local:v
11     t41 = load t40 : int32
12     t42 = add t39, t41 : int32
13
14     t43 = addr_of_symbol local:candidate
15     store t43, t42 : int32
16
17     t44 = load t43 : int32
18     t45 = load t33 : int32
19     t46 = addr_index t32, t45, 4
20     t47 = load t46 : int32
21     t48 = cmp_gt t44, t47 : bool
22     br t48, L13, L14

```

Listing 19: The inner DP update in IR. $t37 = \text{sub } t34, t36$ is $\text{cap} - w$; $t38 = \text{addr_index } t32, t37, 4$ addresses $\text{dp}[\text{cap} - w]$; the later $t46 = \text{addr_index } t32, t45, 4$ addresses $\text{dp}[\text{cap}]$. Both indices are run-time values, so both accesses are bounds-checked.

```

1  129: LOAD_TEMP    t34
2  134: LOAD_TEMP    t36
3  139: SUB
4  140: STORE_TEMP    t37
5  145: LOAD_TEMP    t32
6  150: LOAD_TEMP    t37
7  155: ADDR_INDEX_CHK elem=4 size=11
8  164: STORE_TEMP    t38

```

Listing 20: The $\text{dp}[\text{cap} - w]$ access lowered to bytecode. `LOAD_TEMP t34`, `LOAD_TEMP t36`, `SUB`, `STORE_TEMP t37` compute $\text{cap} - w$; then the base `LOAD_TEMP t32` and index `LOAD_TEMP t37` are pushed and `ADDR_INDEX_CHK elem=4 size=11` consumes both, forming and checking the address in one opcode. The check is baked into the opcode choice, not re-derived at run time.

VM dispatches 4,205 opcodes. Native is only 4.25 times the step count — the smallest expansion in the chapter, and not a coincidence. With no helpers to call and no functions to enter, we are watching iteration in its rawest form. Each IR step compiles to a small, fixed handful of FRISC instructions — a load, an address computation, a bounds comparison, a load or store. Profiled by routine, `F_MAIN` is 100% of the executed instructions — there is nowhere else for the time to go. This is what a program costs when it is all algorithm: a steady four-to-one expansion from IR to machine code, with no multiplier hiding in the runtime.

Key insight

The bounds check is paid per access, decided per program point. A hot loop that indexes the same array a thousand times performs the bounds comparison a thousand times — it is genuine run-time work, one compare-and-branch per access on the FRISC and one `ADDR_INDEX_CHK` on the VM. But the *decision* of whether to check that access was made exactly once, when the lowerer ran the bounds analysis. The interpreter, by contrast, re-runs that analysis every time it enters the function (Section 11.6); for a program like this one, where all the work is in `main`, that is once, but for a hot helper called repeatedly it would be a recurring tax the compiled and VM paths do not pay. Knapsack shows the floor: when nothing else intervenes, iteration plus checking is about four native instructions per IR operation.

Optimisation trims the edges — 1,563 steps fall to 1,468, native 6,643 to 6,473 — as load-forwarding removes repeated reads of the same slot inside the inner loop. The loop structure itself survives, because the optimiser preserves the program’s iteration: it cannot know the table is small without running the program, and it will not unroll a loop whose bound is a run-time value. The savings are real but modest, which is the honest shape of optimisation on a tight, already-lean loop.

Knapsack used only integer addition and comparison, which the FRISC has in hardware. The last two programs reach for arithmetic the FRISC does not have at all — not integer division this time, but fixed-point multiplication — and the cost structure shifts again.

13.4 Horner in fixed point: a multiply in every iteration

One multiply per coefficient: that is what Horner’s method costs in theory, and on the FRISC it is exactly what hurts. We evaluate a four-coefficient polynomial at $x = 1.5$ in floating point (Listing 21), which the compiler represents as `Q16.16` fixed point (Chapter 9). The interpreter runs it in 83 steps and returns 282,624 — the `Q16.16` encoding of 4.3125. The native pipeline takes 820 instructions, and over half of them, we will find, are spent inside one helper routine. The arithmetic trick Horner is famous for is to nest the polynomial as $(\dots((c_0)x + c_1)x + c_2)\dots$ rather than summing $c_0 + c_1x + c_2x^2 + \dots$ term by term, trading a pile of multiplications for one multiply and one add per coefficient. On a target where

each of those minimised multiplies is a function call, minimising them is exactly the right instinct.

```

1  // EXPECT: 4.3125
2  float coeff[4] = {1.5, -2.0, 0.5, 3.0};
3
4  float main(void) {
5      float x = 1.5;
6      float result = coeff[0];
7      int i;
8      for (i = 1; i < 4; i++) {
9          result = result * x + coeff[i];
10     }
11     return result;
12 }
```

Listing 21: Horner’s method over four float coefficients. Each loop iteration performs one fixed-point multiply (`result * x`) and one fixed-point add (`+ coeff[i]`). The result 282,624 is 4.3125 in Q16.16.

The loop body in IR is one `mul` and one `add` on floats (Listing 22). The multiply is the expensive one. A Q16.16 multiply is not a hardware operation even on a processor that *has* integer multiply: it must compute the full 64-bit product and shift right by 16 to restore the fixed-point scale, which on the FRISC means a call to `F_F MUL`. The bytecode names the operation directly — the lowerer, knowing from the IR types that this is a fixed-point multiply, emits a single `MUL_Q16` opcode (Listing 23), having pushed the constant $x = 1.5$ as its Q16.16 encoding 98,304. The type that chose `MUL_Q16` over plain `MUL` has, by then, done its job and vanished.

The full spread falls between the integer-helper-bound GCD and the helper-free knapsack, as you would expect of a program that calls one helper per iteration of a short loop: 820 native instructions against 83 interpreter steps and 223 VM dispatches, about a tenfold native expansion. The helper share is where the opener’s promise comes due. Profiled by routine, `F_F MUL` is 53.4% of the executed instructions — just over half the program’s native cost is the fixed-point multiply helper, called once per loop iteration as the loop folds in coefficients two through four (the first coefficient seeds the accumulator, so four coefficients cost three multiplies). The remaining 46.6% is `main` — the loop counter, the coefficient array accesses, the fixed-point add (which, being an ordinary integer add at the bit level, needs no helper). The story is the same one GCD told, scaled down: a single arithmetic operator the hardware lacks comes to dominate the budget, because each appearance of it is a subroutine, not an instruction.

```

1      L2:
2          t9 = addr_of_symbol local:result
3          t10 = load t9 : float
4          t11 = addr_of_symbol local:x
5          t12 = load t11 : float
6          t13 = mul t10, t12 : float
7
8          t14 = addr_of_symbol global:coeff
9          t15 = addr_of_symbol local:i
10         t16 = load t15 : int32
11         t17 = addr_index t14, t16, 4
12         t18 = load t17 : float
13         t19 = add t13, t18 : float
14         store t9, t19 : float
15         jmp L3

```

Listing 22: One iteration of Horner in IR. `t13 = mul t10, t12 : float` is the fixed-point multiply result $\ast x$; `t19 = add t13, t18 : float` adds the next coefficient `coeff[i]`. The `:` float annotation is what tells the back ends to use fixed-point arithmetic.

```

11      206: LOAD_TEMP    t10
12      211: LOAD_TEMP    t12
13      216: MUL_Q16
14      217: STORE_TEMP    t13

```

Listing 23: The multiply result $\ast x$ lowered to bytecode: two `LOAD_TEMP`s push the operands and `MUL_Q16` — the opcode the lowerer emits for a Q16.16 multiply, chosen from the IR’s float result type — consumes them and pushes the product. Earlier in the function (not shown) the constant $x = 1.5$ was initialised with `PUSH_CONST 98,304`, its fixed-point encoding 1.5×2^{16} .

Key insight

Fixed point makes multiplication a function call. The seductive thing about Q16.16 is that addition and subtraction are just integer addition and subtraction — the scale cancels, and the FRISC’s hardware ADD suffices. Multiplication and division are where the scale must be corrected, and that correction — a widening multiply and a shift — is a helper routine. So in fixed-point code the *adds are free and the multiplies are calls*, and a program’s cost is governed by how many multiplies it does, not how much arithmetic it does overall. Horner is the polynomial-evaluation scheme that does the fewest multiplies; choosing it over the naive term-by-term sum is, on this target, directly a choice about how many F_FMUL calls to make.

What about the loop, though? The loop runs only three times, and its body genuinely depends on a different `coeff[i]` each iteration, so the optimiser cannot fold it — -01 shaves 83 steps to 77 and 820 instructions to 811, and no more. The multiply must happen at run time because its operands are not all known at compile time. But what if they *were*? What if every operand in a fixed-point computation were a literal the compiler could see? Then the entire computation could move from run time to compile time — and the last program shows exactly that happening.

13.5 Kinematics: a program that vanishes

The kinematics formula $x = x_0 + v_0 t + \frac{1}{2} a t^2$ is the straight-line companion to Horner: the same fixed-point multiplies and adds, but no loop, just one long expression (Listing 24). With $x_0 = 0.5$, $v_0 = 1.5$, $a = 1.0$, and $t = 2.0$ it evaluates to 5.5, encoded in Q16.16 as 360,448. At -00 it is the most helper-dominated program in the chapter. At -01 it nearly disappears. Both facts come from the same root, and together they make the deepest point in the chapter.

At -00 the IR is twenty-three instructions of loads, fixed-point multiplies, and adds (Listing 25). Run it and the multiply helper swallows the program: native executes 756 FRISC instructions, the interpreter 23 steps, the VM 69 dispatches — a native expansion of nearly 33×, the steepest in the chapter, because with only 23 IR steps and several of them fixed-point multiplies, the 78.9% share that F_FMUL commands has almost nothing to share with. There is no loop to amortise anything and no integer work to dilute the helpers; the program is essentially a sequence of F_FMUL calls with a little glue. On a per-step basis it is the most expensive program here.

Now turn on -01 and look again (Listing 26). The entire function is one instruction: `ret #5.5:float`. The optimiser’s typed constant folding (Chapter 7) noticed that every operand of every operation is a literal — x_0 , v_0 , a , t are all constants, so $v_0 t$ is a constant, so $x_0 + v_0 t$ is a constant, and so on up the expression tree — and evaluated the whole formula at compile time, in Q16.16, exactly as the runtime would have. The 23 IR instructions collapse to 1. The interpreter now takes a single step. The VM dispatches 2 opcodes. The


```

1  // EXPECT: 5,5
2  // Q16.16: 360448
3
4  float main(void) {
5      float x0 = 0.5;
6      float v0 = 1.5;
7      float a = 1.0;
8      float t = 2.0;
9      float x;
10
11     x = x0 + v0 * t + 0.5 * a * t * t;
12     return x;
13 }

```

Listing 24: Position under constant acceleration, all in fixed point. Every operand is a literal, which is irrelevant to how the formula runs at -00 and decisive at -01. It returns 5.5, encoded as 360,448.

```

16  L0:
17      t0 = addr_of_symbol local:x0
18      store t0, #0.5:float : float
19      t1 = addr_of_symbol local:v0
20      store t1, #1.5:float : float
21      t2 = addr_of_symbol local:a
22      store t2, #1.0:float : float
23      t3 = addr_of_symbol local:t
24      store t3, #2.0:float : float
25
26      t4 = load t0 : float
27      t5 = load t1 : float
28      t6 = load t3 : float
29      t7 = mul t5, t6 : float
30      t8 = add t4, t7 : float
31
32      t9 = load t2 : float
33      t10 = mul #0.5:float, t9 : float
34
35      t11 = load t3 : float
36      t12 = mul t10, t11 : float
37      t13 = mul t12, t11 : float
38
39      t14 = add t8, t13 : float
40
41      t15 = addr_of_symbol local:x
42      store t15, t14 : float
43      t16 = load t15 : float
44      ret t16

```

Listing 25: The unoptimised IR: twenty-three instructions that store the four inputs, load them back, and combine them with fixed-point multiplies (mul ...: float) and adds. Every operand is ultimately a compile-time constant, but -00 does not look.

native code is 35 instructions, almost all of it the unavoidable program bootstrap and main’s prologue and epilogue — the actual “computation” is loading the constant 360,448 into a register and returning it. The fixed-point multiply helper is never called, because there is no multiply left to perform.

```
16     L0:  
17     ret #5.5:float  
18 .endfunc
```

Listing 26: The optimised IR — the entire function. Typed constant folding evaluated the straight-line formula at compile time and replaced twenty-three instructions with the single constant return `ret #5.5:float`. The program’s work moved from run time to compile time.

Key insight

The cost of an operation depends on when it runs. The same fixed-point computation costs 756 native instructions if it runs at execution time and nothing at all if the optimiser runs it at compile time. Kinematics and recursive Fibonacci are the two poles of what the optimiser can do, and it is illuminating to hold them together. Fibonacci’s redundancy was algorithmic and invisible to a local rewrite, so `-00` and `-01` were identical and the optimiser was a bystander. Kinematics’ computation was entirely over constants, so the optimiser could perform it once, ahead of time, and erase it from every execution forever. Most real programs live between these poles — some of the work folds, most does not — but the poles themselves are the clearest statement of what an optimiser is: a machine for moving work from run time to compile time whenever it can prove the move is safe.

This is also the sharpest illustration of why the three back ends must be measured on the *same* IR. All three consumed the folded `-01` IR and all three got cheaper together — the interpreter from 23 steps to 1, the VM from 69 dispatches to 2, the native code from 756 instructions to 35. None of them did the folding; the optimiser did, before any of them ran, and they simply inherited a shorter program. The optimiser neither knows nor cares which back end will execute its output, which is exactly why a single optimisation pass pays off three times over. With all five programs measured, we can finally lay the numbers side by side and read what the portfolio says as a whole.

13.6 What the three pipelines told us

Here is the single fact the whole portfolio comes down to: the bytecode VM’s expansion factor barely moves, while the native one swings eightfold. Across all five programs — recursion, integer helpers, pure iteration, fixed-point multiply — the VM dispatches between 2.69 and 3.00 opcodes per interpreter step, while the native pipeline runs anywhere from

4.25 to 32.9 instructions per step on the very same programs. Table 13.1 collects every measurement behind that contrast in one place: the five programs, the three back ends, at both optimisation levels. Read down a column and you see one back end’s spread across programs; read across a row and you see one program run three ways. Both directions carry a lesson.

Table 13.1: Executed-operation counts for the five case-study programs, run three ways at both optimisation levels. “FRISC” is executed machine instructions, “Interp” is interpreter steps, “VM” is bytecode dispatches. Results are return values; the fixed-point programs return Q16.16 encodings ($282,624 = 4.3125$, $360,448 = 5.5$). All three back ends agree on every result, at both levels.

Program	Result	-00			-01		
		FRISC	Interp	VM	FRISC	Interp	VM
recursive Fibonacci	6765	1,368,194	218,909	623,888	1,368,194	218,909	623,888
GCD / LCM	420	1,457	72	194	1,404	57	153
knapsack DP	15	6,643	1,563	4,205	6,473	1,468	4,035
Horner (fixed-point)	282,624	820	83	223	811	77	214
kinematics	360,448	756	23	69	35	1	2

The headline figure (Figure 13.2) plots the -00 counts on a logarithmic axis, because the programs span five orders of magnitude — recursive Fibonacci does more than a million native instructions, kinematics fewer than a thousand. On any single program the three back ends rank the same way: the native pipeline does the most work, the interpreter the least, the VM in between. That ranking is structural and never inverts, and it pays to understand why.

The interpreter does the least because its step is the *coarsest* unit: one whole three-address IR instruction per step, with the host processor underneath performing every multiply, divide, and remainder in hardware. The VM does more because each IR instruction lowered to roughly 2.7 stack opcodes, the expansion we measured in Chapter 12; its dispatch is finer-grained but cheaper. The native pipeline does the most because it expands each IR instruction into real machine instructions *and* emulates in software every operator the FRISC lacks. But the ranking by work-count is emphatically not a ranking by speed: the interpreter’s 218,909 steps and the simulator’s 1.37 million instructions run on different machines, and on real hardware the native code — with no interpreter overhead per step — would be the fastest of the three by a wide margin. The counts measure where the work is, not how long it takes.

Figure 13.3 is where that through-line finally resolves into a cause. The VM’s near-constant factor is a property of the lowering — the fixed number of stack opcodes a three-address instruction becomes — and so it does not depend on what the program computes. The native factor, laid out program by program, is anything but constant: $4.25\times$ for knapsack, $6.25\times$ for Fibonacci, $9.9\times$ for Horner, $20.2\times$ for GCD, $32.9\times$ for kinematics. That entire spread is explained by two things and only two: how many calls the program makes, and

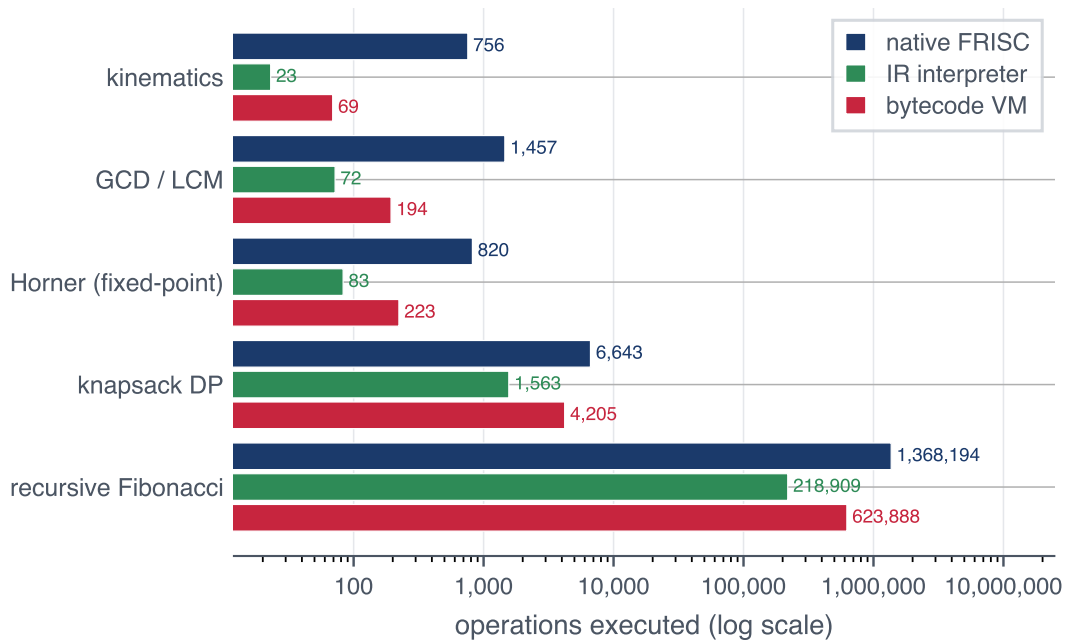


Figure 13.2: Executed operations per back end at `-O0`, on a logarithmic axis. Native FRISC always does the most work, the interpreter the least, the VM between — a structural ordering, not an accident of any one program. The five-order span from recursive Fibonacci to kinematics is why the axis is logarithmic.

how many of its operations are software-emulated helpers. The VM’s cost is intrinsic to the translation; the native cost is intrinsic to the *target*.

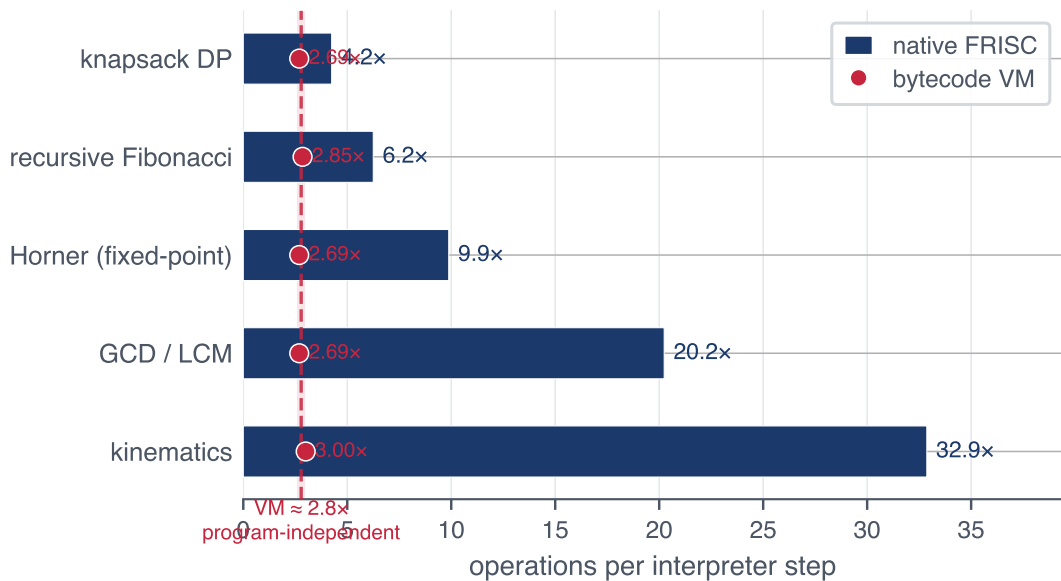


Figure 13.3: Operations executed per interpreter step, at $-O0$. The bytecode VM’s expansion is flat near 2.7 across all programs — a property of the IR-to-stack lowering. The native FRISC expansion varies eightfold, from $4.25\times$ to $32.9\times$, driven entirely by call overhead and software-emulated arithmetic helpers.

The helper budget (Figure 13.4) makes the second of those two forces visible directly. Two programs — Fibonacci and knapsack — spend nothing on arithmetic helpers: Fibonacci’s cost is all call overhead, knapsack’s all iteration. The other three spend the majority of their native instructions inside helper routines: 53% for Horner, 76% for GCD, 78.9% for kinematics. When a program’s bar is mostly accent-coloured, its performance is governed less by its algorithm than by the gap between what the source language assumes — multiply, divide, fixed-point math — and what the FRISC provides.

Three conclusions survive the portfolio, and they are the reason the chapter exists. The first is that the three back ends never disagreed on an answer — thirty runs, five programs, two optimisation levels, one value each. Three independent readers of the IR, built by different means at different times, triangulated every result; the differential oracle of Chapter 11 and Chapter 12 is not a theoretical nicety but a property we relied on to trust every number in Table 13.1. The second is that cost is structured, not arbitrary: interpreter steps count the algorithm’s essential operations, VM dispatches multiply that by a fixed lowering constant near 2.7, and native instructions multiply it by a program-dependent factor that is entirely about calls and helpers. Knowing those three numbers, you can predict any cell of the table

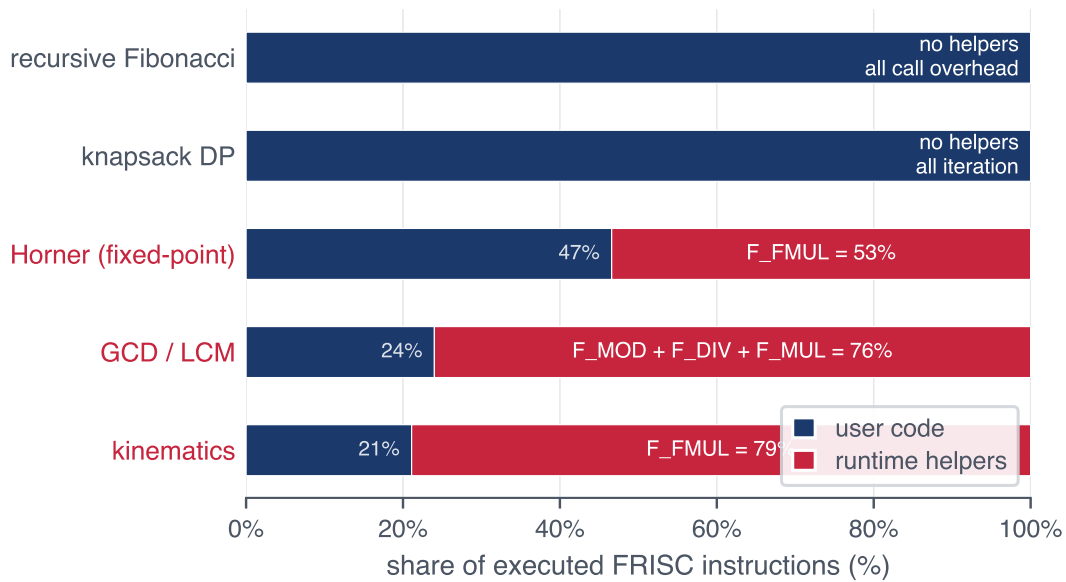


Figure 13.4: Share of executed FRISC instructions spent in user code versus runtime helper routines, at -O0. Two programs call no helpers at all; three spend over half their native cost emulating arithmetic the ISA lacks. The helper share, not the algorithm, decides these programs' native cost.

from any other. The third is that optimisation’s leverage is wildly uneven — nil on naive recursion, total on a constant straight-line formula — and that it always helps all three back ends at once, because all three consume the IR the optimiser shrinks.

Key insight

The IR is the contract; the target sets the price. The single representation that made all of this measurable is the typed IR. It is what the optimiser rewrites, what all three back ends consume, and what lets one program be compared three ways without ambiguity about *which* program. The interpreter’s step count is the purest reading of that IR’s intrinsic work. Everything above that count — the VM’s $2.7\times$, the native code’s $4\text{-to-}33\times$ — is the price of a particular way of running it on a particular machine. Separating the contract from the price is what an intermediate representation is *for*, and it is why every measurement in this chapter could be taken against the same artifact.

What the portfolio does not tell us is how these costs add up across many programs at once, which passes of the optimiser earn their keep, or how the cycle budget is distributed when the suite is large. Those are questions about the compiler’s aggregate behaviour rather than any single program’s anatomy, and they are the subject of [Chapter 14](#), which runs a wider suite and measures the optimiser pass by pass. And the one cost we found we could not reach from the source — the helper share baked into the instruction set — points squarely at [Chapter 15](#), where a different target, a register-based VM, or a hardware multiplier would each move a number this chapter could only report.

Practice

Exercise 13.1 (Applied · ★★★)

Breadth versus depth. [Section 13.1](#) distinguishes a recursion’s *breadth* (total calls, the work) from its *depth* (longest live chain, the stack). Using the recurrence for the call count, $C(n) = 1 + C(n-1) + C(n-2)$ with $C(0) = C(1) = 1$, verify that $C(20) = 21,891$. Then give the depth of the recursion for `fib(20)`, and explain why doubling the depth (to `fib(40)`) multiplies the breadth by far more than two.

Exercise 13.2 (Applied · ★★★)

Reading the helper budget. [Section 13.2](#) reports that `F_MOD`, `F_DIV`, and `F_MUL` together account for 76% of GCD’s 1,457 executed instructions. From the per-routine shares given (51.9%, 19.9%, 4.3%), compute how many instructions each helper executed. Given that `F_MOD` runs a 32-iteration loop, estimate how many times the GCD program called `F_MOD`, and check your estimate against the fact that Euclid on 84 and 30 takes three modulo steps (the loop body runs three times: $84 \bmod 30$, $30 \bmod 24$,

24 mod 6) before y reaches zero.

Exercise 13.3 (Applied · ★★★)

Predicting a cell. Section 13.6 claims you can predict any cell of Table 13.1 from the interpreter step count plus two factors. For knapsack at -00, the interpreter takes 1,563 steps. Using the VM expansion of ≈ 2.7 and the native expansion of 4.25 given in the text, predict the VM dispatch count and the native instruction count, and compare with the table's 4,205 and 6,643. Why is the VM prediction more accurate than the native one across programs?

Exercise 13.4 (Applied · ★★★)

When folding wins. Kinematics folds completely at -01 (23 steps to 1) while recursive Fibonacci does not change at all. State the precise property of each program that explains the difference. Then construct a small variant of the kinematics program that the optimiser can *not* fully fold — by making exactly one operand a value read from outside the function — and predict, without running it, whether its -01 interpreter step count will be closer to 1 or to 23.

Exercise 13.5 (Applied · ★★★)

A fairer speed question. The chapter is careful to call its counts a measure of work, not of time, and notes that the native code runs on a simulator. Suppose you wanted to make a defensible wall-clock comparison between the three back ends. Describe what you would have to hold constant, why timing the JavaScript simulator against the JVM interpreter would be misleading, and what measurement *would* fairly capture the native pipeline's intended advantage. Relate your answer to the distinction the chapter draws between counting work and measuring speed.

Project

Exercise 13.6 (Lab · ★★★)

Project: profile your own three back ends. You now have, from the projects of Chapter 11 and Chapter 12, a tinyC tree-walking interpreter and a bytecode VM, alongside the reference compiled path. This project asks you to build the measurement harness this chapter used and turn it on your own programs.

Open `ch13-casestudies/starter/` in the companion repository, where two reference back ends — a tree-walking interpreter and a stack bytecode VM, the miniatures of Chapter 11 and Chapter 12 — are provided complete, each reporting its work to a listener. Four `// TODO: implement markers` await you in `Profiler.java`. The first

two count interpreter steps and VM dispatches respectively, by listening to the two engines — the two currencies this chapter reports. The third is a native-cost estimator: walk the same execution and accumulate, for each IR operation, how many FRISC instructions it would expand into — a few for an add, a software loop for a multiply or divide — so you can fill the native column without a full back end. (tinyC has no FRISC code generator yet, so the companion estimates native cost with a documented model rather than stepping a real simulator as [Chapter 10](#) did.) The fourth buckets that native cost by the routine each instruction ran in, and totals the helper share, reproducing [Figure 13.4](#)’s budget.

Run all three back ends over the five int-only programs the module ships — recursive Fibonacci, Euclid’s GCD with a least-common-multiple wrapper, a counting loop, an integer power, and a Horner polynomial evaluated by scaling integers by hand — and reproduce the three-way table and the expansion factors. (tinyC is int-only, so the book’s array-based knapsack and its genuinely-float programs are not expressible; the integer programs carry the same call-bound, helper-bound, and iteration-bound stories.) Then add one program of your own designed to make the native expansion as *large* as you can (hint: a multiply-heavy inner loop) and one to make it as *small* as you can (hint: pure integer iteration), and explain the factors you measured.

Your turn

Open `frisccc-companion/ch13-casestudies/starter/` and build the measurement harness the Project describes: the step-and-dispatch counters around the provided interpreter and VM, the native-cost estimator that expands each IR operation, and the per-routine profiler that produces a helper budget. When it runs, `Report` prints this chapter’s table for all five programs, and you can read any tinyC program’s cost three ways — as interpreter steps, as VM dispatches, and as estimated native instructions — and watch the expansion factors of this chapter appear in your own numbers: the VM’s flat $\approx 2.7\times$ per step, and the native expansion that swings with how many helpers and calls the program makes. The reference harness waits in `frisccc-companion/ch13-casestudies/solution/`.

Notes and further reading

The discipline of comparing implementations by counting primitive operations rather than wall-clock time — and of being honest about what each count does and does not measure — is older than computing itself, but Knuth’s analyses in *The Art of Computer Programming* [50] are its enduring model: count the operations that matter, prove the count, and only then worry about the constant factor of a particular machine. The fixed-point arithmetic that dominated three of our five programs, and the reasons multiplication and division are the expensive operations while addition is free, are treated thoroughly in the same work’s volume on seminumerical algorithms [51]. The dynamic-programming structure of

the knapsack solver — the table, the in-place update, the computed indices that forced our bounds checks — is the textbook treatment in Cormen and colleagues [19].

The gap this chapter kept running into — between operations the source language assumes and operations the target provides in hardware — is the central concern of computer architecture, and Hennessy and Patterson [35] is the place to understand why a hardware multiplier or divider costs what it costs and when an instruction set chooses to omit one, as the FRISC does. Their account of the reduced-instruction-set philosophy [68] illuminates exactly the trade our helper routines pay for: a smaller, simpler instruction set bought at the price of emulating in software what a richer set would do in silicon. For the compiler's side of that same gap — how a back end decides what to emit for an operation the target lacks, and how an optimiser like ours moves work from run time to compile time — Cooper and Torczon [18] and Muchnick [62] are the standard references, the latter especially on the analyses that let constant folding fire as completely as it did on our kinematics formula.

The next chapter widens the lens. Where this one anatomised five programs one at a time, Chapter 14 runs a larger suite, measures the optimiser pass by pass to see which rewrites actually earn their place, and asks where the cycle budget goes when you average over many programs rather than dwelling on one. The case studies gave us the vocabulary — steps, dispatches, instructions, expansion factors, helper budgets — and the performance chapter puts it to work at scale.

Chapter 14

Performance

“It is a capital mistake to theorise before one has data.”
— Arthur Conan Doyle, *A Scandal in Bohemia*

The previous chapter put five programs under a microscope, one at a time, and watched where each one spent its work. This chapter pulls back to the wide angle. We have a compiler with an optimiser of sixteen passes, an `-O0` path that emits whatever the lowering walker produced and an `-O1` path that rewrites it, and a back end that turns the result into FRISC. The natural questions, now that all the machinery exists, are the ones a compiler writer cannot answer by reading the code: does `-O1` actually make programs run less work, and if so by how much; which of the sixteen passes earn their place and which merely tidy; where does the cycle budget go once we stop staring at one program and average over many; and how many times does the pipeline have to sweep its passes before the IR stops changing. Each of these is a measurement, and the whole chapter is an exercise in measuring carefully and reporting honestly — including when the honest answer is “less than you would hope.”

We promised, at the end of [Chapter 13](#), to widen the lens this way, and [Chapter 7](#) left two explicit IOUs that come due here: the “full ablation methodology” behind its per-pass figure, and the empirical count of pipeline iterations its fixpoint loop actually takes. We pay both debts below. What we will *not* do is reach for a stopwatch, and the first section explains why a chapter titled “Performance” measures everything except elapsed time.

14.1 What to measure, and what we are not measuring

Mytkowicz and colleagues once took a fixed binary and made it measure faster or slower than itself — not by changing a line of code, but by changing the order of a few unrelated environment variables and the link order of the object files [63]. The shift they could conjure out of nothing was larger than the speed-up a careful optimisation typically earns. That is the cautionary tale behind this chapter’s one firm methodological choice: we do not reach for a clock.

The obvious way to find out whether `-O1` makes a program faster is, after all, to compile it both ways, run each, and time them. We decline, and not out of squeamishness about benchmarking but the opposite — a wish to measure something that will still be true on a machine we have never seen. Consider what a wall-clock number would actually contain on

our stack. A FRISCcc-compiled program does not run on FRISC silicon; it runs on `friscjs`, a JavaScript simulator, executed by whatever Node build and laptop happens to be on the desk. The IR interpreter and the bytecode VM of [Chapter 11](#) and [Chapter 12](#) run on the JVM, whose just-in-time compiler may or may not have warmed up, whose garbage collector may or may not pause mid-measurement, and whose scheduler shares the core with a browser and a build daemon. A millisecond reading off that tower tells us about the tower far more than about the compiler — and, as Mytkowicz showed, a number an unrelated environment variable can flip is not a number we want to print in a book.

So we count instead of time. Every quantity in this chapter is a *count of primitive operations*, and counts have the cardinal virtue of being deterministic: the same program compiled the same way executes exactly the same number of FRISC instructions on your machine as on ours, today and in ten years, because the count is a property of the program and the compiler, not of the host. We lean on four such counts, and it is worth being precise about each because they answer different questions.

Four counts, four questions. The *executed FRISC instruction count* is the dynamic one: how many fetch–decode–execute cycles the simulator performs from `CALL F_MAIN` to `HALT`, obtained by stepping it one instruction at a time exactly as the runner of [Chapter 10](#) does. It answers “how much work does this program do when it runs?” The *static IR-instruction count* and *static FRISC instruction count* are size measures — how many lines of IR, and how many machine instructions, the compiler emits — and they answer “how big is the program?”, which is a different question with, as we are about to see, a surprisingly different answer. The *sweep count* is the optimiser’s own: how many times the fixpoint pipeline runs its list of passes before a sweep changes nothing. Dynamic work, static size, and convergence are three independent axes, and conflating them is the easiest way to draw a confident wrong conclusion about an optimiser.

The portfolio we measure is fourteen programs, listed in [Table 14.1](#). It is built outward from the five case studies of [Chapter 13](#) — so the numbers there and here can be read together — and the two anchor programs the book has carried throughout, then widened with seven more drawn from the `examples/real_world/` corpus to span the cost structures the compiler has to cope with: tight integer loops, naive recursion, array iteration with bounds checks, fixed-point and floating-point arithmetic that leans on the runtime helpers, and bitwise code that exercises the shift passes. None was chosen to flatter the optimiser.

Every number below was produced by a single reproducible harness, `PerfHarness`, which drives the real front end and back end — the same `IrPipeline` and `FrisccCodeGenerator` the production compiler uses — so that nothing here is a special-cased re-implementation that might quietly disagree with the compiler the rest of the book describes. The harness adds exactly one capability the command-line compiler lacks: it can run the `-O1` pass list with any single pass removed, which is what the ablation of [Section 14.3](#) needs. The harness and its counting tools are collected in [Section 14.6](#); for now all we need from it is the one

Program	Domain	What it stresses
Fibonacci (iter)	math	counted loop; front-end anchor
Fibonacci (rec)	math	naive recursion; call/return overhead
prime sieve	real	nested loops, array stores; back-end anchor
GCD / LCM	math	integer F_MOD and F_DIV
integer sqrt	math	integer F_DIV in a loop
knapsack DP	real	array iteration, bounds checks, no helpers
dot product	real	array traversal, integer F_MUL
quicksort (max)	real	recursion over an array, no helpers
CRC checksum	real	F_MOD-bound bitwise reduction
activity sel.	eng	greedy scan, comparisons
Horner (fixed)	math	Q16.16 multiply-accumulate, F_FML
kinematics	physics	straight-line float; folds to a constant
k-means 1D	ml	float means, F_DIV
projectile	physics	float step loop, F_FML

Table 14.1: The fourteen-program suite, built from the five case studies of [Chapter 13](#) (Fibonacci rec, GCD/LCM, knapsack DP, Horner, kinematics), the book’s two anchor programs (Fibonacci iter and the prime sieve), and seven more drawn to widen the coverage of cost structures. Sources live under `examples/real_world/`.

question it lets us ask, and ask honestly — when -O1 is switched on, how much work does it actually save?

14.2 What -O1 buys

Here is the headline, and it is not the one the marketing copy for an optimiser would write. Summed across the whole suite, turning on -O1 reduces the number of executed FRISC instructions by *two-tenths of one percent* — from 1,547,690 down to 1,544,567, a saving of 3,123 instructions out of a million and a half. Over the same suite, -O1 shrinks the static IR by 10.5% and the static FRISC code by 6%. The optimiser makes the program markedly smaller and barely makes it do less work. [Figure 14.1](#) shows both facets side by side, and the gap between the two panels is the most important thing in the chapter.

How can an optimiser delete a tenth of the code and next to none of the work? Because the two measures count different things. The static count is one tally per instruction in the program text; the dynamic count weights each instruction by how many times it executes. The passes of [Chapter 7](#) are very good at removing instructions that the lowering walker emitted redundantly — a copy that is never read, a re-loaded address that was already in a register, a constant computed twice. But those redundancies sit mostly in straight-line setup code that runs once, not in the bodies of the hot loops where the executed-instruction count actually accumulates. Delete ten instructions that each run once and you save ten executed instructions; the loop that runs forty thousand times is unchanged because the optimiser found nothing in it to remove. It is the same gap a profiler teaches every programmer the

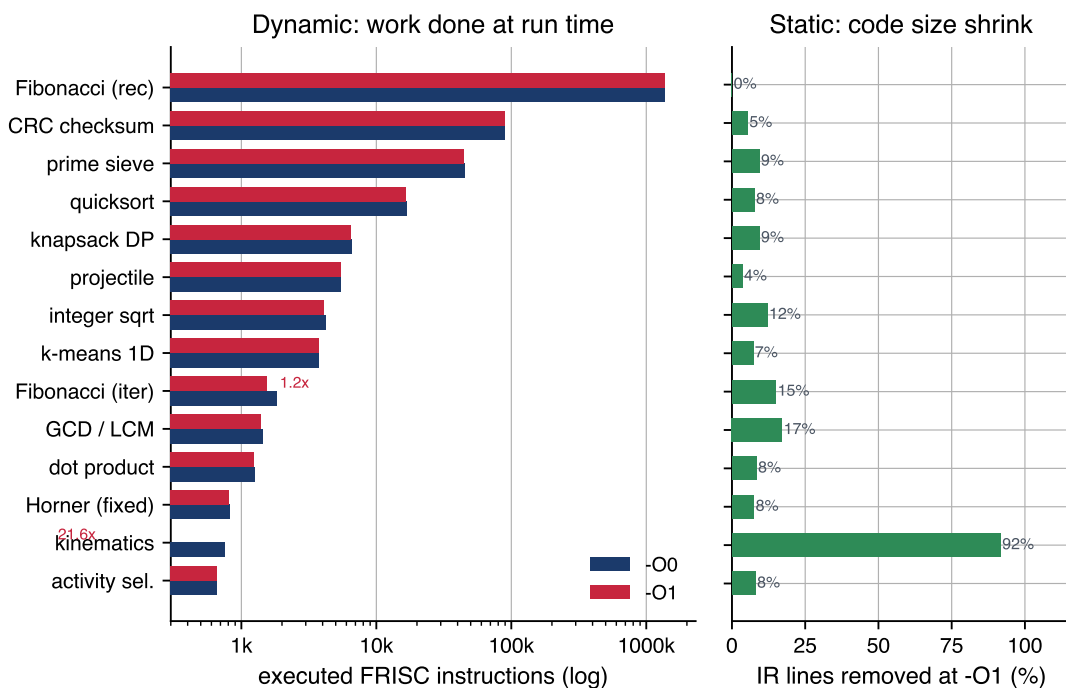


Figure 14.1: What -O1 buys across the suite. *Left:* executed FRISC instructions at -O0 (dark) and -O1 (red), on a log axis spanning three orders of magnitude. For all but one program the two bars are nearly the same length — -O1 changes the work done at run time hardly at all. *Right:* the percentage of IR lines -O1 removes, which is substantial almost everywhere. Smaller is not the same as faster.

first time they open one: how many instructions a function *contains* tells you almost nothing about how much of the running time it *owns*. The static win is real and the dynamic win is real; they are just concentrated in different parts of the program.

Key insight

Smaller is not faster. A code-size reduction and a work reduction are different measurements and an optimiser can deliver one without the other. Our -O1 shrinks IR by about a tenth across the suite while cutting executed work by a fifth of a percent, because the instructions it removes run rarely and the instructions that run often it cannot remove. An optimiser aimed at speed must find the hot path; one aimed at size need not. Reporting only one number — “ten percent smaller” — would be true and misleading at once.

The suite aggregate, though, hides as much as it reveals, because it is dominated by its largest term. Recursive Fibonacci alone executes 1,368,194 instructions — more than the other thirteen programs combined — and -O1 changes that number not at all: the optimiser has nothing to say about a naive recurrence, every instruction in the call sequence is live, and -O0 and -O1 produce byte-identical machine code. With one unoptimisable elephant sitting on the scale, the suite-wide dynamic saving was almost guaranteed to be tiny regardless of how well the optimiser did on the other thirteen. Table 14.2 gives the per-program numbers, and they tell the fuller story.

Program	executed instr.			IR lines		
	-O0	-O1	ratio	-O0	-O1	Δ%
Fibonacci (rec)	1,368,194	1,368,194	1.00	22	22	0
CRC checksum	89,936	89,828	1.00	75	71	5
prime sieve	45,671	44,355	1.03	118	107	9
quicksort (max)	16,989	16,696	1.02	118	109	8
knapsack DP	6,643	6,473	1.03	107	97	9
projectile	5,509	5,509	1.00	57	55	4
integer sqrt	4,216	4,084	1.03	58	51	12
k-means 1D	3,752	3,742	1.00	166	154	7
Fibonacci (iter)	1,823	1,543	1.18	40	34	15
GCD / LCM	1,457	1,404	1.04	41	34	17
dot product	1,259	1,235	1.02	36	33	8
Horner (fixed)	820	811	1.01	40	37	8
kinematics	756	35	21.6	24	2	92
activity sel.	665	658	1.01	50	46	8

Table 14.2: Per-program -O0 versus -O1, sorted by executed-instruction count. The *ratio* column is the dynamic speed-up (executed instructions at -O0 divided by -O1); the *Δ%* column is the static IR-line reduction. Note how the two right-hand groups disagree: IR shrinks 8–17% on programs whose executed work barely moves.

One program breaks the pattern completely, and it is worth dwelling on because it shows the optimiser at the top of its game. The kinematics formula is straight-line floating-point arithmetic — no loop, no input, every operand a literal — and at -O1 the constant folder of [Chapter 7](#) evaluates the entire computation at compile time and the function collapses to a single `ret #5.5`. Its executed-instruction count falls from 756 to 35, a $21.6\times$ reduction, and its IR shrinks by 92%. This is the case where the static and dynamic wins coincide, and they coincide precisely because the thing the optimiser removed — the whole arithmetic body — was also the thing that executed. When the redundancy is on the hot path, deleting it is both a size win and a speed win. The lesson of the suite is that this alignment is the exception; on thirteen of fourteen programs the optimiser’s removals and the program’s hot path live in different places.

14.3 Per-pass ablation: which rewrites earn their place

Remove loop-invariant code motion from the optimiser and recompile the whole suite, and the number of FRISC instructions the suite executes does not change — not by one. Yet the same pass, [Chapter 7](#) showed, removes several lines of IR from every program that has a loop. Both facts are true at once, and the space between them is what this section is about.

That earlier chapter ranked the passes with a histogram of IR lines removed and deferred the methodology to here. The histogram measured *size*: with one pass disabled, how many more lines of IR survive? This section asks the harder and more useful question, the one about *work*: with one pass disabled, how many more instructions does the compiled program actually *execute*? A pass that deletes twenty lines of run-once setup looks impressive on the size histogram and contributes nothing here; a pass that removes one instruction from the body of a hot loop barely registers on the size histogram and can dominate this one. The two views of the same sixteen passes are genuinely different, and reading them together is the point.

The procedure is the ablation that [Chapter 7](#) promised, and [Algorithm 14.1](#) states it precisely. We establish a baseline by compiling every program at full -O1 and recording its executed-instruction count. Then, for each of the sixteen distinct passes in turn, we rebuild the -O1 pipeline with every occurrence of that one pass removed, recompile all fourteen programs through the real back end, and re-count. The difference between the ablated count and the baseline, summed over the suite, is how much work that pass was responsible for removing. A pass whose absence costs nothing did no dynamic work on this suite, however much IR it deletes.

Two design choices in that procedure deserve a word, because a careless ablation produces numbers that look authoritative and mean nothing. First, we delete *every* occurrence of the pass, not just one: the -O1 list runs dead-temp elimination, control-flow simplification, and unreachable-block elimination twice each (once mid-pipeline, once after strength reduction, as [Chapter 7](#) explained), and leaving one copy in would measure almost nothing for those passes. Second, we re-run the whole fixpoint loop after the deletion rather than splicing the pass out of a single sweep, because the passes interact: removing copy propagation

Algorithm 14.1: Per-pass dynamic ablation: remove one pass, recompile, and count the extra work its absence causes.

Input: programs P ; the -O1 pass list Π (with repeats)

Output: for each distinct pass π , the extra executed instructions its removal costs

```

1 for  $p \in P$  do
2    $b_p \leftarrow$  executed instructions of  $p$  compiled with full  $\Pi$ 
3 for  $\pi \in \text{distinct}(\Pi)$  do
4    $\Pi' \leftarrow \Pi$  with every occurrence of  $\pi$  deleted
5    $\text{extra}_\pi \leftarrow 0$ ;  $\text{affected}_\pi \leftarrow 0$ 
6   for  $p \in P$  do
7      $c \leftarrow$  executed instructions of  $p$  compiled with  $\Pi'$ 
8      $\delta \leftarrow c - b_p$ 
9      $\text{extra}_\pi += \delta$ 
10    if  $\delta \neq 0$  then
11       $\text{affected}_\pi += 1$ 

```

does not merely subtract copy propagation’s own work, it denies dead-temp elimination the copies it would otherwise have found dead. This is the logic of a controlled experiment, where removing one factor exposes not only its own effect but every downstream effect that depended on it; an ablation that switched each pass off in isolation, as though the sixteen were independent columns in a spreadsheet, would undercount every pass whose real value is the opportunity it hands to the next. Ablation measures a pass *together with* the downstream opportunities it creates, which is the honest thing to measure — a pass is worth keeping exactly when the pipeline runs worse without it, interactions and all.

The result, [Figure 14.2](#), splits the sixteen passes cleanly in two. Eight of them change the suite’s executed-instruction count when removed; the other eight change it by exactly nothing. That is not a defect — a pass can be indispensable for code size, or for keeping the IR in a canonical shape the next pass expects, or for correctness, while having no effect on the instructions that actually run — but it is a sharp reminder that the work an optimiser does and the work it *saves at run time* are different ledgers.

[Table 14.3](#) lists the eight passes that pull their weight dynamically. Control-flow simplification leads by a wide margin and by the broadest reach: removing it costs the suite 1,657 executed instructions across nine of the fourteen programs, because the jumps it straightens and the empty blocks it folds are jumps and blocks that would otherwise be *executed*, once per loop iteration, on every program with a loop. Copy propagation and load forwarding follow, each touching five programs — the two passes that keep a value in a register instead of reloading or re-copying it, and the saving compounds inside loops. Then a tail: common-subexpression elimination and dead-store and dead-temp elimination, each shaving a few hundred instructions off a handful of programs.

The *programs* column is as telling as the instruction count. Global value propagation and

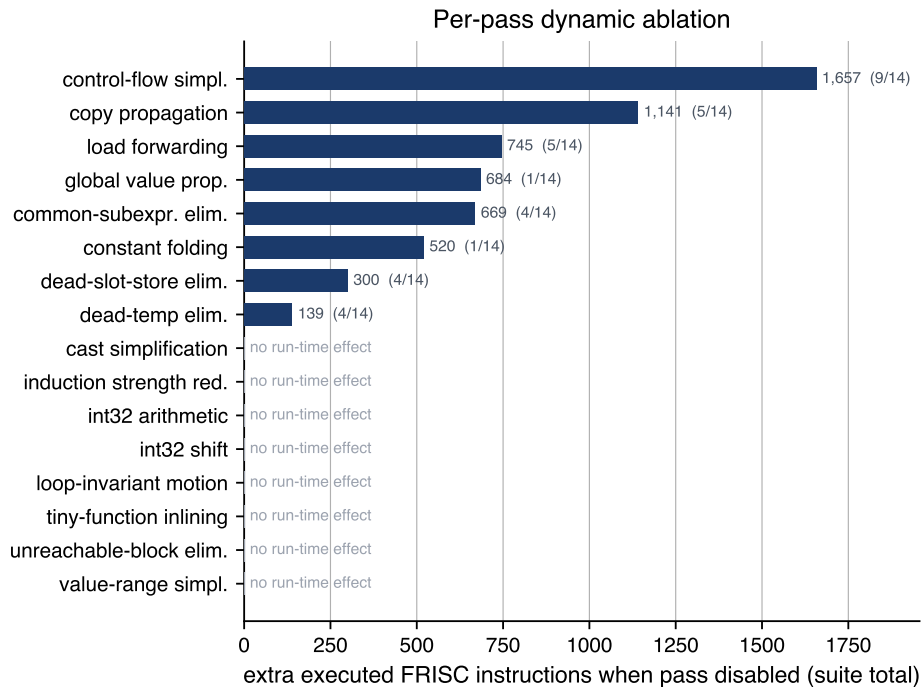


Figure 14.2: Per-pass dynamic ablation. Each bar is the extra executed FRISC instructions the suite performs when that one pass is removed from -O1, summed over all fourteen programs; the count beside each bar is how many of the fourteen programs were affected at all. A short bar means the pass earns its place on size or correctness grounds rather than by removing run-time work.

Pass	extra instr.	programs
control-flow simplification	1,657	9
copy propagation	1,141	5
load forwarding	745	5
global value propagation	684	1
common-subexpression elimination	669	4
typed constant folding	520	1
dead-slot-store elimination	300	4
dead-temp elimination	139	4
<i>No run-time effect on this suite: cast simplification, int32 arithmetic, int32 shift, value-range simplification, loop-invariant motion, induction strength reduction, tiny-function inlining, unreachable-block elimination.</i>		

Table 14.3: The eight passes whose removal changes executed work, with the suite-summed cost of removing each and the number of the fourteen programs affected. The remaining eight passes change no executed-instruction count on this suite.

typed constant folding each move a great deal of work — 684 and 520 instructions — but each on a single program, the one whose constants happen to line up for them; constant folding’s one program is kinematics, the formula that collapses to 5.5. These are passes that do nothing on most programs and everything on the rare program built to suit them. Control-flow simplification is the opposite: a small, steady saving spread across most of the suite. An optimiser needs both kinds, and a single ranking number would have hidden the distinction.

The eight passes with no dynamic effect are the chapter’s sharpest collision with [Chapter 7](#). That chapter’s static histogram ranked the passes by IR lines removed, and its leaders were load forwarding, copy propagation, and constant folding; loop-invariant code motion sat comfortably mid-table, removing a few lines on every program with a loop. Here loop-invariant motion removes *zero* executed instructions, and so does induction strength reduction — the two textbook hot-loop optimisations, the ones whose entire reason for existing is to make loops cheaper. The explanation is not that they failed to fire; [Chapter 7](#) showed them firing. It is that what they hoisted or strength-reduced on these particular loops was either already executed once outside the loop, or was an address computation the back end folds into an addressing mode at no extra instruction, so moving it changed the program’s size without changing the instructions it runs. The lesson is uncomfortable and worth sitting with: a pass can be a famous optimisation, can fire exactly as designed, can shrink the IR — and still save no run-time work on the programs you actually compile. Only the ablation tells you which of your passes are in that position, and only a dynamic ablation at that; the static one of [Chapter 7](#) would have reported LICM as a contributor.

Key insight

Ablate dynamically, or you will keep the wrong passes. Ranking optimisation passes by how much IR they delete rewards passes that tidy run-once code and flatters passes whose wins never reach the hot path. The only ranking that reflects speed is the one that recompiles with each pass removed and counts the instructions that actually execute. On our suite the two rankings disagree sharply — control-flow simplification rises from the middle of the size table to the top of the work table, and loop-invariant motion falls from a steady size contributor to no dynamic effect at all. Measure the thing you care about, not its proxy.

The ablation accounts for every instruction the optimiser can remove. It says nothing, though, about the instructions it *cannot* touch — the ones that belong not to the compiler’s output at all but to the instruction set it was forced to target. On several of our programs those are the majority, and they are where we turn next.

14.4 The helper-cycle budget

[Chapter 13](#) measured, for five programs one at a time, how much of their work went into the runtime helper routines — the software `F_MUL`, `F_DIV`, `F_MOD`, and the fixed-point `F_FML`

that stand in for arithmetic the FRISC has no hardware for. Here we ask the same question of the whole suite at once, at -O1, and the aggregate answer is a trap worth springing on purpose.

Summed across all fourteen programs, 5.7% of executed instructions run inside helper routines — an execution-weighted average that not one individual program actually sits near. Turn on -O1 and the figure is still 5.7% — the optimiser moves the helper budget essentially not at all. A reader who saw only that pair of numbers would conclude that helpers are a rounding error and the compiler is powerless over them, and both halves of that conclusion would be wrong. Figure 14.3 shows why: the per-program helper share runs from zero to 85.7%, and the 5.7% aggregate is low only because the two longest-running programs — recursive Fibonacci and quicksort — use no helpers at all and drown out the rest by sheer instruction volume.

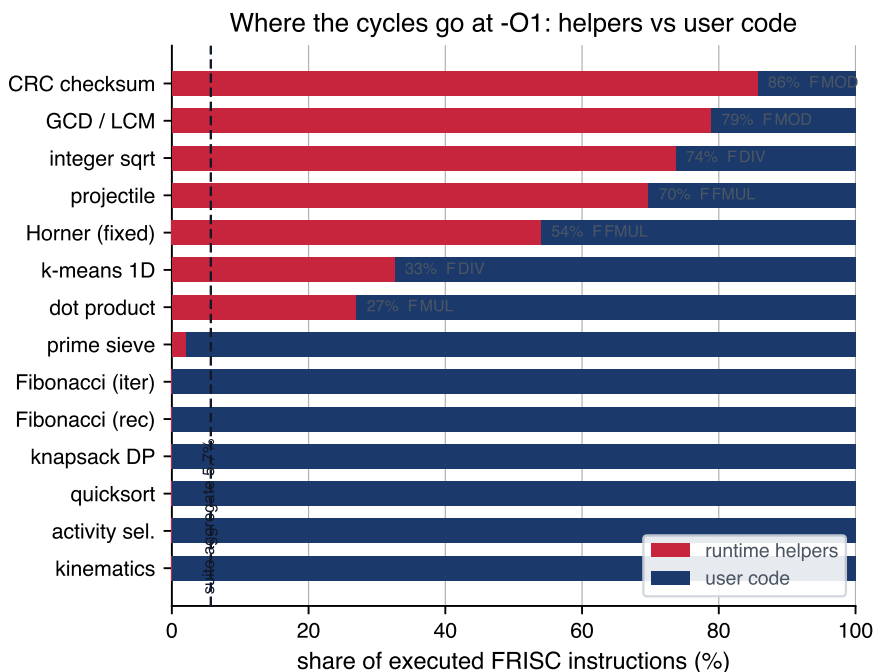


Figure 14.3: Share of executed instructions spent in runtime helpers, per program, at -O1, with the dominant helper labelled. The dashed line is the execution-weighted suite aggregate, 5.7% — a single number the 0–85.7% spread almost entirely conceals. The arithmetic-bound programs at the top spend most of their lives emulating one missing instruction.

The programs at the top of that figure are the ones whose cost is dictated not by the compiler but by the instruction set. CRC checksum spends 85.7% of its instructions in F_MOD; GCD/LCM, 78.9% in the same routine; integer square root, 73.7% in F_DIV; the projectile and Horner loops, 69.7% and 54% respectively in the fixed-point F_FMUL. For

these programs the dominant cost is a single operation the source language assumes and the hardware lacks, and no amount of dead-temp elimination touches it. This is the same gap [Chapter 9](#) built the helpers to bridge, seen now from the performance side: a reduced instruction set buys simplicity in silicon and charges for it in software every time a program multiplies or divides [[35](#), [68](#)].

These programs are Amdahl's law made visible. The argument Gene Amdahl made in 1967 [[5](#)] is that if a fraction f of a program's work cannot be sped up, then no improvement to the rest can push the overall speed-up past $1/(1 - f)$, however heroic. For CRC checksum, f is the 0.857 of its instructions trapped in `F_MOD`, so even a compiler that drove the *other* 14.3% to nothing could not make the program more than about $7\times$ faster. Our optimiser is not working near that ceiling; it is nibbling at the small unshaded slice while the helper owns the rest. The only lever that moves f itself is the one the compiler does not hold — the instruction set.

Key insight

The optimiser cannot remove work the ISA imposes. On the arithmetic-bound programs, the majority of executed instructions live inside a helper routine that emulates a missing hardware operation. The optimiser can sometimes delete the *call* — if the operands fold to a constant, as in kinematics, the multiply never happens — but it cannot make a multiply that genuinely runs cost less than the helper costs. Past a point, the way to make these programs faster is not a cleverer compiler but a different target, and recognising which kind of cost you face is the first step of any optimisation effort.

There is one apparent counter-example, and it confirms the rule. Kinematics spent 79% of its `-00` instructions in `F_FMUL`, as [Chapter 13](#) found — it is a wall of floating-point multiplies — yet at `-01` its helper share is zero. The optimiser did not make the multiplies cheaper; it proved they did not need to happen, folded the whole formula to 5.5, and emitted code that calls no helper because it computes nothing. Eliminating the call is exactly the move the optimiser *can* make, and it only worked because every operand was a compile-time constant. Change one operand to a runtime input and the folding stops cold and the helper share springs back to four-fifths.

The helper budget, then, is set by the instruction set, and the one variable the compiler still controls is how hard it works to reach a fixed point before giving up. That effort has a cost of its own, paid in compile time rather than run time, and it is the last thing this chapter measures.

14.5 Convergence: how many sweeps until the IR stabilises

The last measurement is the optimiser's own internal one. [Chapter 7](#) described the pipeline as a fixpoint loop: run the whole list of passes top to bottom, and if any pass reported a change, run the whole list again, stopping when a complete sweep changes nothing or when

a configured cap of five sweeps is reached. It argued the loop must terminate — every pass either shrinks a well-ordered measure of the IR or leaves it untouched — and it asserted, without showing the data, that “three or four iterations is enough” for the anchor program. This is the same convergence structure as the classic dataflow analyses of [Chapter 7](#): iterate until nothing changes, with a bound to keep the pathological case from running forever — only here a “change” is whether any pass rewrote an instruction, rather than whether a lattice value moved. Now we can show the data, for all fourteen programs, and it sharpens the earlier claim in two directions.

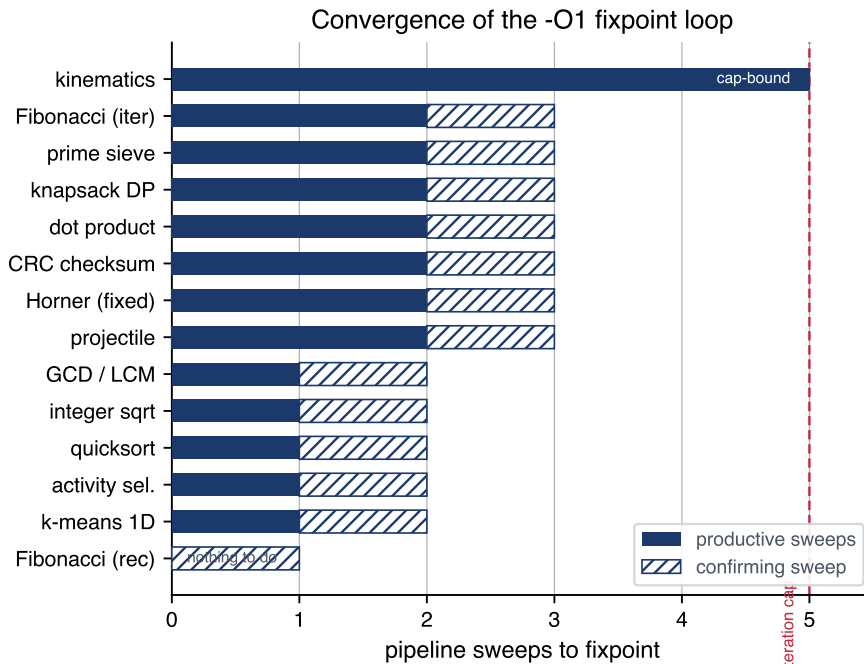


Figure 14.4: Sweeps to a fixpoint, per program. The solid bar counts *productive* sweeps (a sweep that changed the IR); the hatched segment is the single confirming sweep that found nothing left to do and so proved a fixpoint was reached. Recursive Fibonacci needs zero productive sweeps. Kinematics is still changing at the cap of five, so its bar is bounded by the iteration limit, not by a confirmed fixpoint.

The common case, [Figure 14.4](#) shows, is gentle. Seven of the fourteen programs reach a fixpoint after two productive sweeps and a third confirming sweep; five more settle after a single productive sweep. The pattern matches the pipeline’s design: most of the work happens on the first pass through the list, the second sweep cleans up the opportunities the first one exposed, and by the third there is nothing left, which is exactly why the default cap of five is comfortable headroom rather than a binding constraint. Recursive Fibonacci sits at the floor with zero productive sweeps — the first sweep changes nothing, the loop exits immediately, and this is the mechanical reason its `-O0` and `-O1` code were identical

back in [Section 14.2](#).

One program, though, is still changing when the cap stops it, and honesty requires we say so plainly.

When the cap binds. Kinematics runs all five sweeps and reports a change on every one of them. Unlike the other thirteen, it never gets the confirming no-change sweep that proves a fixpoint, so we cannot claim from this data that it *reached* one — only that the answer it produces by sweep five is correct and agrees with `-00`. What drives the churn is the constant folder cascading through a long chain of dependent multiplies: each sweep folds one more layer of the expression and propagates the result, exposing the next layer for the sweep after. The chain is finite, so the loop would terminate on its own, but it is deep enough to outlast a cap chosen for the common case. This is the “rare adversarial case” [Chapter 7](#) mentioned the cap exists to bound, caught in the act — and it is a useful reminder that an iteration limit is a safety valve, not a correctness mechanism: the IR is well-formed and the result is right at every sweep, capped or not.

The convergence data also closes the loop on a compile-time worry. The pipeline re-validates the IR before and after the outer loop, and a program that needed many sweeps would pay for each one. Thirteen of our fourteen programs run the pass list at most three times; the optimiser’s own cost is bounded by a small constant on everything except the one deeply-foldable outlier, and even there by the cap. Whatever else `-01` is, it is not a compiler that grinds.

14.6 Reproducing these numbers

Every figure and table in this chapter regenerates from a clean compiler with one command. The driver is `book/scripts/ch14_bench.sh`, which knows the fourteen-program suite and emits the five data files the figures read (`ch14_suite.dat`, `ch14_ablation.dat`, `ch14_helper_share.dat`, `ch14_helper_perprog.dat`, `ch14_convergence.dat`). It calls two small Node counters over the assembled FRISC — `frisc_count.js`, which steps the simulator one `performCycle` at a time exactly as the runner does, and `frisc_profile.js`, which buckets executed instructions by enclosing routine to separate helper cycles from user cycles — and the Java `PerfHarness` that builds the ablated pipelines. No step uses a clock, so a re-run on any machine reproduces the counts to the instruction. The full build recipe is in [Appendix E](#).

Practice

Exercise 14.1 (Applied · ★★☆☆)

The weighted-mean trap. The suite-wide dynamic saving of `-O1` is 0.2%, but recursive Fibonacci alone contributes 1,368,194 of the suite’s 1,547,690 executed instructions. Recompute the suite-wide saving with recursive Fibonacci excluded. Why do the two numbers differ so much, and which one would you quote to someone deciding whether to enable `-O1`?

Exercise 14.2 (Conceptual · ★☆☆)

Static versus dynamic. Table 14.2 shows GCD/LCM with the largest IR-line reduction of any program whose dynamic ratio stays near 1.00 (17%, setting aside the kinematics outlier that folds away entirely) and a dynamic ratio of only 1.04. Explain, in terms of where its instructions execute, how a pass can remove a sixth of the IR while removing only a twenty-fifth of the work.

Exercise 14.3 (Applied · ★★☆☆)

Reading the ablation. Pick any pass that Figure 14.2 reports as having no run-time effect on the suite. Using its description in Chapter 7, give one concrete reason it nonetheless earns its place in the pipeline — on code size, on canonical form for a later pass, or on correctness.

Exercise 14.4 (Applied · ★★☆☆)

Helper arithmetic. CRC checksum spends 85.7% of its instructions in `F_MOD`. Suppose a future FRISC revision added a hardware modulo instruction costing four cycles. Estimate the program’s new executed-instruction count and its speed-up, stating the assumption you make about the current per-call cost of `F_MOD`. Compare your speed-up to the Amdahl ceiling for $f = 0.857$.

Exercise 14.5 (Conceptual · ★☆☆)

Convergence by hand. For a program that needs two productive sweeps, argue why the optimiser cannot in general predict that number in advance and must instead discover it by running the loop until a sweep is unproductive. What property of the pass set guarantees the loop stops at all?

Project

Exercise 14.6 (Lab · ★★★)

Project: measure your own optimiser. The companion repository’s `ch14-performance` module turns the apparatus of this chapter on the `tinyC` compiler you have been building. The starter project in `frisc-cc-companion/ch14-performance/starter/` gives you a skeleton `PerfHarness`, the fourteen-program corpus translated into `tinyC`, and stub implementations of an executed-operation counter and an ablation loop, each with a `// TODO: implement` marker.

Your job has four parts, each mirroring a section of this chapter. First, implement the counter — whatever your back end’s natural unit of work is — and reproduce the `-00-versus-01` table for the corpus, so you can see for yourself how far your static and dynamic savings diverge. Second, add the ablation loop of [Algorithm 14.1](#): rebuild the `-01` pipeline with each pass removed in turn, recount, and report the per-pass dynamic impact. Third, instrument your pipeline’s fixpoint loop to report productive and confirming sweeps, and then find or construct an input that makes your own iteration cap bind, the way kinematics does ours. Finally, write a one-paragraph honest summary: does `-01` make your programs smaller, faster, or both, and by how much? Quote the weighted aggregate *and* the per-program spread, because — as this chapter laboured to show — a single number for either is almost always misleading.

When the counter and the ablation loop are done, run

```
mvn test -pl ch14-performance
```

from the companion root. The test suite checks that your counter is deterministic — the same program counted twice yields the same number — and that your ablation baseline reproduces a full `-01` run exactly. The reference build is in `ch14-performance/solution/`. *Stretch goal:* extend the harness to combine the per-program ratios with the geometric mean as well as the execution-weighted arithmetic mean of [Section 14.2](#), and explain in two sentences which summary you trust and why.

Your turn

Open `frisc-cc-companion/ch14-performance/starter/` and implement the executed-operation counter and the ablation loop, then run `mvn test -pl ch14-performance` from the companion root. When the determinism and ablation-baseline tests pass, you will have measured your own optimiser the way we measured ours — and you will know, rather than hope, which of your passes earn their place at run time and which only make the code look smaller.

Notes and further reading

The decision to count operations rather than measure elapsed time is not timidity; it is the lesson of a literature on how easily wall-clock measurement misleads. Mytkowicz and colleagues [63] demonstrate measurement bias large enough to reverse a benchmark’s verdict

from causes as innocent as link order and environment size, and their paper is the one to read before trusting any runtime number. The discipline of counting the operations that matter and proving the count, rather than timing a particular machine, is Knuth's throughout *The Art of Computer Programming* [50]; we follow it here for the same reason he does, that a count outlives the machine it was taken on.

Summarising a benchmark suite in a single number is its own minefield. The 0.2% aggregate of Section 14.2 is an execution-weighted arithmetic mean, dominated by its largest term; Fleming and Wallace [28] make the classic argument that ratios across a suite should be combined with the geometric mean precisely so that no single program dominates, and re-deriving our headline that way is instructive. The deeper principle behind the helper-budget section — that the speed-up available from improving one part of a program is capped by the fraction of time spent there — is Amdahl's [5]; the arithmetic-bound programs of Figure 14.3 are Amdahl's law made visible, with the helper routine as the part the compiler cannot speed up.

For benchmark suite design and the question of what makes a program representative, the SPEC CPU methodology [36] is the industrial reference; our fourteen programs are a teaching-scale analogue, chosen for disjoint cost structures rather than statistical coverage. And for the standard treatment of the analyses that make ablation meaningful — why removing one pass changes what the others can do — Muchnick [62] and Cooper and Torczon [18] remain the references, the latter especially good on the pass-ordering interactions that Section 14.3 measures rather than merely describes.

Chapter 15

Where to take it next

“The best way to predict the future is to invent it.”
— Alan Kay

We opened this book with a black box. A C program went in one end, an executable came out the other, and the transformation in between was something almost none of us could give a faithful account of — “the program was characters; then it wasn’t,” as [Chapter 1](#) put it. Fourteen chapters later the box is open. Every phase has been named, motivated, built, measured, and in the case of the optimiser’s most delicate passes, proved correct. There is no longer a mysterious middle. There is a lexer, a parser, a type checker, a lowering walker, an optimiser, a code generator, a runtime, and a simulator, and we have watched a program move through all of them and come out the far side as the integer it always was.

This final chapter does two things, and they pull in opposite directions on purpose. The first half looks *back*: it assembles the whole machine on a single page, so that the phases we studied one at a time can be seen at once as the system they always formed, and so the arc of the book closes where it began. The second half looks *forward*: it takes the questions the earlier chapters deliberately left open — what a register-based virtual machine would change, what an SSA-form intermediate representation would buy the optimiser, what a just-in-time compiler would add on top of the bytecode VM, and what it would take to point the back end at a different machine or the front end at a different language — and sketches each honestly enough that you could start building it tomorrow.

That is the right note to end on, because the point of reading a compiler from cover to cover was never to memorise FRISCcc. It was to earn the ability to change it, port it, or replace it with one of your own. So the chapter is less a conclusion than a handoff. We will recap what we built, draw an honest map of what we left out, walk the most interesting roads not taken, and then — in the spirit of the epigraph — get out of the way.

15.1 The whole machine

Here is everything, on one page. [Figure 15.1](#) is the diagram [Chapter 1](#) could not have shown you, because at that point none of its boxes had been opened. Now they all have. Read it top to bottom and it is the journey of a program; read it as a shape and it is the architecture of the book.

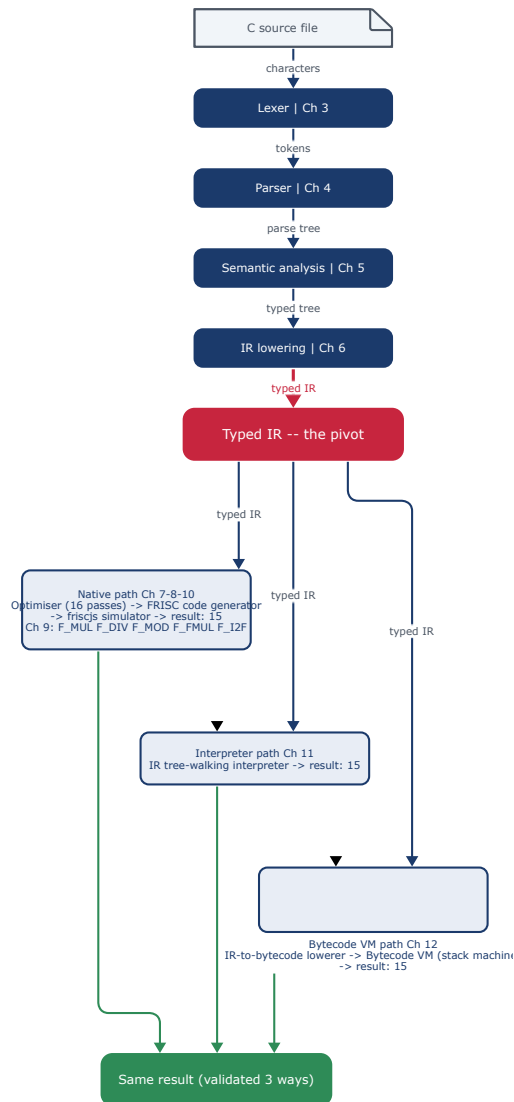


Figure 15.1: The complete FRISCCc system. The front-end spine runs down the top: source characters become tokens (Chapter 3), tokens become a parse tree (Chapter 4), the tree acquires types (Chapter 5), and the typed tree is lowered to the intermediate representation (Chapter 6). The IR is the pivot of the whole design — everything above produces it, and three independent back ends below consume it: the native path of optimiser (Chapter 7), code generator (Chapter 8), and friscjs simulator (Chapter 10), drawing on the runtime helpers (Chapter 9); the tree-walking interpreter (Chapter 11); and the bytecode VM (Chapter 12). All three converge on the same answer — the three-way agreement Chapter 13 leaned on.

Trace the spine first. A source file is a string of bytes, and the *lexer* of Chapter 3 chops it into tokens by running a battery of finite-state machines in parallel and taking the longest match at every step — the maximal-munch discipline we compiled out of regular expressions via Thompson construction and subset construction. The *parser* of Chapter 4 consumes that token stream and, driven by an LR(1) table built from the grammar, shifts and reduces its way to a parse tree, tracking nested structure that no regular language ever could. The *semantic analyser* of Chapter 5 walks that tree, builds the symbol table, infers a type for every expression, and refuses the programs that the grammar was too permissive to reject; it is where `int x; x();` finally meets an objection. And the *lowering walker* of Chapter 6 flattens the verified tree into the typed, three-address intermediate representation that is the hinge of the entire compiler.

Key insight

The IR is the waist of the hourglass. Everything in the front end exists to produce the intermediate representation; everything in the back end exists to consume it. That single design decision — introduced in Chapter 6 as another level of indirection — is what lets one book carry *three* back ends without three compilers. The front end was written once. The native code generator, the tree-walking interpreter, and the bytecode VM each read the same IR and never once looked at a token, a parse tree, or a type. Decouple the two halves at a well-designed interface and each half becomes replaceable on its own.

Below the IR the path forks three ways, and that fork is the quiet thesis of the whole book. The path the first ten chapters built is the native one: the *optimiser* of Chapter 7 runs its sixteen passes to a fixpoint, rewriting the IR into a leaner IR that computes the same values; the *code generator* of Chapter 8 lowers that IR to FRISC assembly, realising every abstraction the front end relied on — frames, the calling convention, addressing modes — as concrete instructions; the *runtime* of Chapter 9 supplies the helper routines (`F_MUL`, `F_DIV`, `F_MOD`, the fixed-point `F_FML`) that stand in for arithmetic the reduced instruction set lacks; and the *simulator* of Chapter 10 runs the result one fetch–decode–execute cycle at a time until it halts. But Chapter 11 and Chapter 12 showed the IR is not married to that path. A tree-walking interpreter can evaluate the IR directly, no code generation at all, trading speed for radical simplicity. A bytecode VM can lower the IR one step further to a flat stream of stack-machine opcodes and dispatch them in a tight loop. Three readings of one IR — and, as Chapter 12 insisted, three *independent* readings, which is why any disagreement between them was a bug report. We validated that agreement across all the example programs, and Chapter 13 put five of them on stage to run three ways and arrive, every time, at the same number.

Part V was where the machine stopped being a thing we built and became a thing we measured. Chapter 13 took five programs apart one at a time and found their costs lived in startlingly different places — calls for recursive Fibonacci, the software `F_MOD` for GCD, pure iteration for the knapsack table. Chapter 14 pulled back to a fourteen-program suite and reported the honest, uncomfortable aggregate: `-O1` shrinks the static IR by about a tenth while cutting *executed* work by two-tenths of one percent, because the redundancies the

optimiser removes sit in run-once setup code and the hot loops it cannot touch are where the cycles actually go. Smaller is not faster; the only way to know which passes earn their place at run time is to ablate them dynamically and count; and on the arithmetic-bound programs the majority of the cycles are trapped in a helper emulating one instruction the hardware does not have — a cost the compiler cannot reach because it does not live in the program at all.

That last finding is the bridge to the rest of this chapter. Three of the four open questions the book deferred to here are, at bottom, the same question asked from different sides: *what do you change when the compiler has done all it can and the cost lives somewhere the compiler cannot reach?* Sometimes the answer is a better intermediate representation, sometimes a different execution engine, and sometimes a different machine entirely. We take them one at a time — but first, the honest map of what is missing.

15.2 An honest list of what we left out

FRISCcc is complete in the sense that every program it accepts runs correctly, end to end, on real artifacts we can regenerate from a clean build. It is emphatically not complete in the sense of being a compiler you would ship. The subset of C it accepts was chosen to be small enough to understand on the page, and the choices that bought that understandability are exactly the choices a production compiler does not get to make. It is worth laying them out plainly, because an honest inventory of what is missing is the most useful map of where to go next. [Figure 15.2](#) groups the gaps by which part of the pipeline each one would perturb.

The source language is where the omissions are most visible to a user. FRISCcc has no `struct` and no pointers, which means no linked data structures, no dynamic allocation, and no aliasing — and the absence of aliasing, not coincidentally, is what let the load-forwarding and dead-store passes of [Chapter 7](#) reason about memory as simply as they did. Add pointers and the optimiser’s comfortable assumption that two memory accesses through different names cannot overlap collapses, and you inherit the entire literature of alias analysis. The language’s `float` is a fixed-point Q16.16 quantity emulated by the helper routines of [Chapter 9](#), not an IEEE-754 floating-point number; real floats would mean either hardware the FRISC does not have or a far larger and more careful set of helpers, and a host of rounding questions the fixed-point representation lets us duck. None of these is an oversight. Each is a deliberate trade of generality for clarity, and each is a well-marked road out of the teaching subset into the language people actually use.

The middle of the pipeline — the IR and the optimiser — is where the most *interesting* extensions live, because they change not what the compiler accepts but how well it reasons. The single largest such change would be to rebuild the IR in static single assignment form, and it earns its own section below. The back end’s omissions are more prosaic but no less real: FRISCcc assigns every IR temporary a stack slot and reloads it on every use, leaning on the load-forwarding pass to paper over the inefficiency, where a production compiler would run a proper graph-colouring register allocator and keep hot values in registers across a whole loop. And the entire back end is welded to one target. The $M + N$ argument of

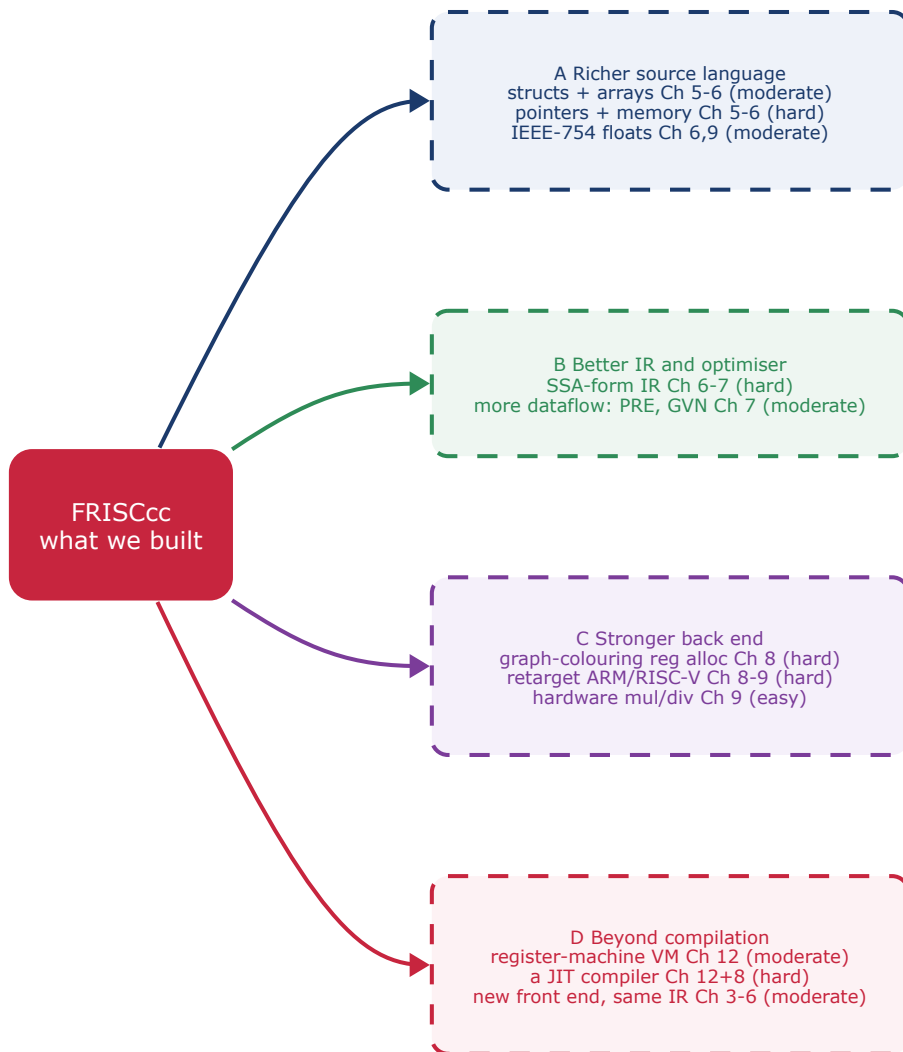


Figure 15.2: The extension horizon. At the centre is the compiler we built; radiating out are the candidate extensions, grouped by which part of the pipeline each one would change and tagged with the chapters whose machinery it touches and a rough difficulty. A richer source language reaches back into the front end; a better IR and optimiser sit at the waist; a stronger back end and a different target live on the native path; and the register VM and JIT extend Part IV. The cheapest win of all — a FRISC with a hardware multiplier — is a change to the *machine*, not the compiler.

Chapter 1 — that an IR turns the problem of M languages times N machines into M front ends plus N back ends — is only a promise until you write the second back end or the second front end and collect on it.

We will not work through all of these. We will take the four the book explicitly promised to return to, in order of how far each reaches into the machine: a different VM, a different IR, a different execution model, and a different target. Each is a real project of a few hundred lines on top of what we have. The first to close is the one Chapter 12 named twice and deferred both times: what changes if the bytecode VM keeps its operands in named registers instead of an implicit stack?

15.3 A register machine of our own

Chapter 12 built a *stack* machine and, twice, pointed here for the alternative. On the stack VM, instructions do not name their operands; they take them from the top of an operand stack and leave their results there. That choice bought the smallest, simplest instructions in the field — an ADD is a single opcode that pops two and pushes one, and the lowerer that produces it is almost trivial. The cost is *traffic*: every value a computation touches has to be pushed before it can be used and popped after, so a single three-address IR instruction becomes several stack instructions, and the VM’s dispatch loop — the most expensive thing it does — runs once for each of them.

A *register* machine makes the opposite trade, and Figure 15.3 puts the two side by side on one IR line. Instead of an implicit stack, a register VM gives each instruction explicit operand fields, exactly as the three-address IR already does. The IR instruction `t8 = add t5, t7` becomes a single bytecode instruction `ADD t8, t5, t7`, carrying three operand indices in its encoding. One dispatch instead of four. The instruction is larger — it has to name three operands where the stack opcode named none — but the VM crosses its dispatch loop a quarter as often.

Which is faster is not a matter of taste; it was settled empirically. Shi and colleagues took the same VM and built it both ways, and found that register bytecode executes *substantially fewer* instructions — at the price of each instruction being larger and the bytecode file bigger overall [74]. Because the dominant cost in an interpreter is the dispatch itself — the indirect jump from one opcode’s handler to the next, which a modern processor’s branch predictor finds hard to predict [26] — doing fewer, fatter dispatches is the trade that pays. This is why the production VMs that came after the JVM’s stack design, notably Lua’s and Android’s Dalvik, are register machines.

What would rebuilding our VM that way actually touch? Less than you might fear, and the reason is the IR. Our IR is *already* a register machine: every instruction already names its operands and its destination. The stack VM of Chapter 12 earned its keep by *flattening* that structure away into pushes and pops; a register VM would simply keep it. The lowerer would get simpler, not more complex — it maps each three-address IR instruction to one bytecode instruction rather than expanding it — and the temporaries `t0`, `t1`, `t2` would

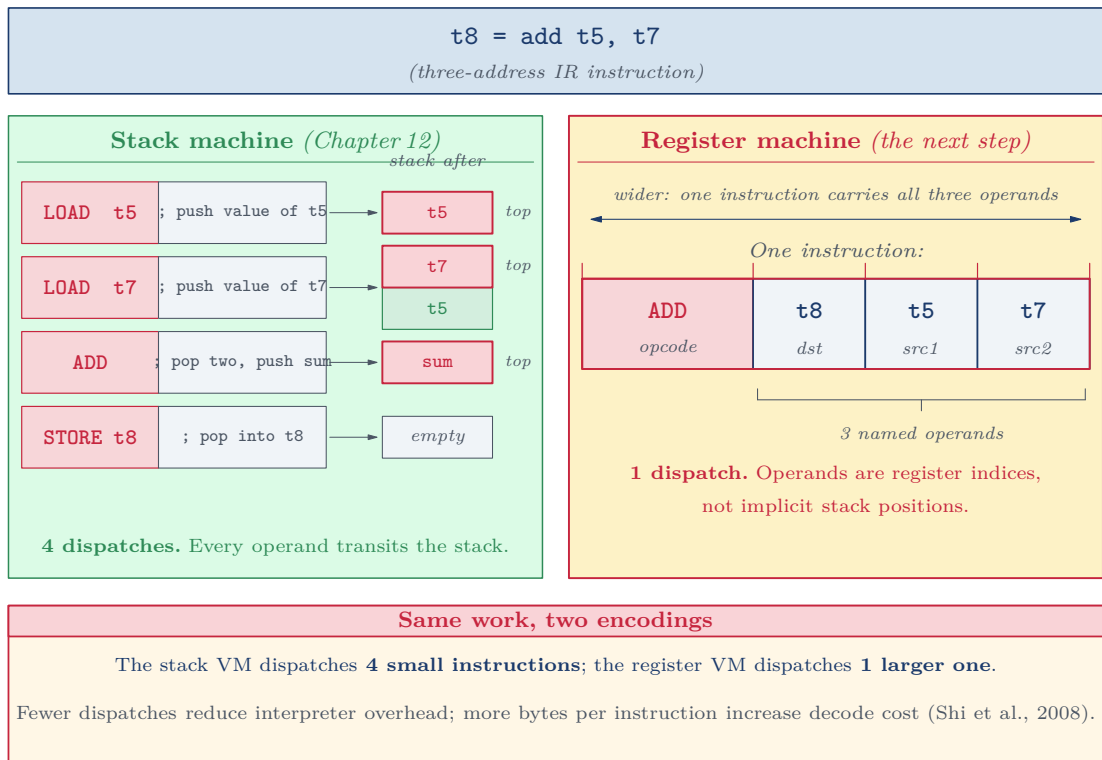


Figure 15.3: The same IR instruction, `t8 = add t5, t7`, lowered two ways. *Left*: the stack machine of Chapter 12 needs four opcodes — load, load, add, store — and the operands flow implicitly through the operand stack shown evolving alongside. *Right*: a register machine encodes the whole computation as one instruction with three explicit operand fields. Fewer, larger instructions: the central trade-off Shi and colleagues measured [74].

become virtual-register indices directly. What grows is the VM core: each handler now decodes operand fields instead of touching a stack, the instruction format is wider and variable, and you need a convention for how many virtual registers a frame may hold. None of it disturbs anything above the IR. The front end would not know the difference, which is the whole point of having built the front end once.

Key insight

The stack/register choice is a VM-local decision. Because both designs consume the same three-address IR, rebuilding the bytecode VM from a stack machine into a register machine changes only the lowerer and the dispatch loop — two files in Part IV. Nothing in the front end, the type system, or the optimiser is even aware of it. That is the dividend of the IR boundary, paid out exactly where [Chapter 1](#) promised it would be.

15.4 SSA, and what it would change about [Chapter 7](#)

Take the smallest program that gives an optimiser trouble: a variable assigned on both branches of an if and read afterwards. In FRISCCc's IR that variable has two definitions under one name, and the reader after the merge cannot statically know which branch ran. So before copy propagation can substitute its value, common-subexpression elimination can recognise it, or load forwarding can keep it in a register, each pass must first answer the same question — *which definition reaches here?* It answers by scanning the block, separately, every time. That question — reaching definitions — is the recurring cost of the whole optimiser, and the shape of the IR is what forces it to be asked again and again.

Static single assignment form, usually just called SSA, removes the question by making its answer unique. The rule is exactly what the name says: every variable is assigned exactly once. The two branches get different names — x1 on one, x2 on the other — and at the merge, where the value's identity depends on a road not yet known, a special pseudo-instruction reconciles them the way a merge lane reconciles two on-ramps: it describes the car that comes out without needing to know which ramp it took. That instruction is the *phi-node*, and [Figure 15.4](#) shows it on the smallest control-flow graph that needs one.

The phi-node looks like a curiosity and is in fact the entire point. Once it is in place, every use in the program refers to exactly one definition, by name, with no analysis required: x3 is defined in exactly one place, full stop. Copy propagation becomes a one-line rewrite — if t9 = t5 and each name has a single definition, every use of t9 can be replaced by t5 unconditionally, because t5 cannot have been reassigned. Common-subexpression elimination becomes a hash-table lookup on the unique definitions. The reaching-definitions analysis that several passes of [Chapter 7](#) each performed separately is computed *once*, structurally, when the program is converted into SSA, and then every pass reads it for free. This is why essentially every serious optimising compiler built since the early 1990s — GCC, LLVM [\[53\]](#), the JVM's just-in-time compilers — uses SSA as its workhorse representation.

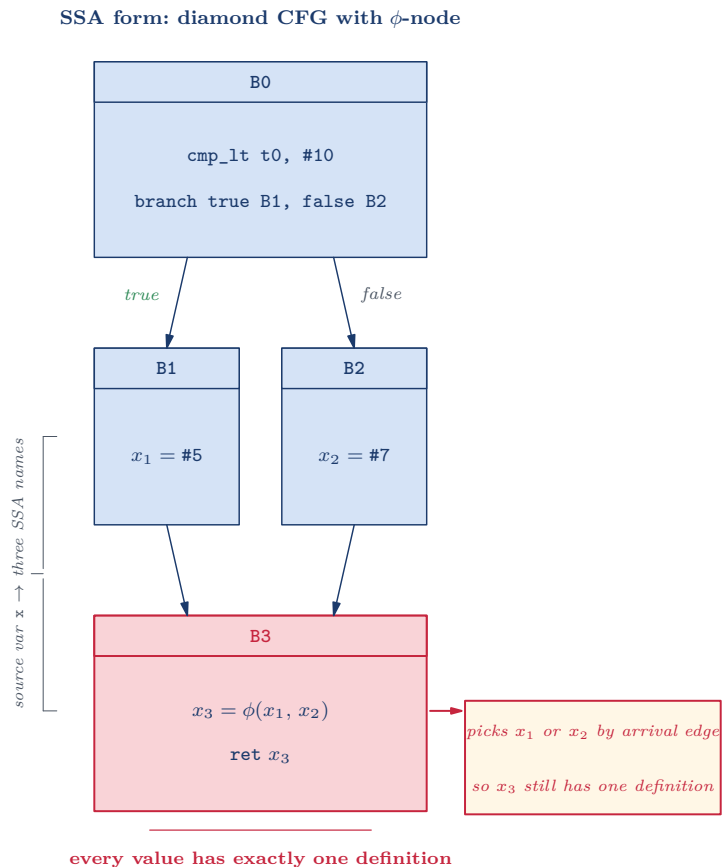


Figure 15.4: A variable assigned on both branches of a conditional, in SSA form. The two assignments are renamed x_1 and x_2 , each now the single definition of its name. At the join, the phi-node $x_3 = \phi(x_1, x_2)$ selects whichever value flowed in along the edge actually taken, so x_3 too has exactly one definition. Every value in the program now has a single, statically known definition — which is precisely the fact the optimiser of Chapter 7 had to recompute by hand.

The conversion is not free, and honesty requires saying where the cost moved rather than vanished. Placing phi-nodes in the right blocks, and no more blocks than necessary, is the one genuinely subtle algorithm in the whole business; the standard solution computes the *dominance frontier* of each definition and is the contribution of Cytron and colleagues' 1991 paper [21], which builds on the fast dominator algorithm of Lengauer and Tarjan [54]; together they are the reference everyone still starts from. And SSA is a representation for *analysis*, not for execution — no real machine has a phi instruction — so before code generation you must convert *out* of SSA, replacing each phi with ordinary copies on the incoming edges, a step that interacts in genuinely tricky ways with the register allocator. So SSA does not remove work from the compiler so much as relocate it: out of every individual pass, and into two well-studied conversions at the IR's edges. For a compiler with sixteen passes that each pay the reaching-definitions tax, that is a trade worth making, and rebuilding FRISCCc's IR around SSA is the single change that would most improve the optimiser of Chapter 7 without touching a line of the front end.

Why we did not build it. SSA would have made Chapter 7 a better optimiser and a worse first chapter on optimisation. The dominance-frontier algorithm, the phi-placement proof, and the out-of-SSA copy-insertion problem are three substantial topics that have nothing to do with *what* an optimisation does and everything to do with the representation it runs on. Teaching constant folding, dead-code elimination, and loop-invariant motion on a plain three-address IR let those passes be understood on their own terms first. SSA is the natural second course, and now that the first is finished, it is the one to take.

Rebuilding the IR around SSA is the largest change this compiler could make to how it *reasons*. The road we take next changes something else entirely — not how the compiler thinks, but when it runs.

15.5 What a JIT would change

The bytecode VM of Chapter 12 stopped one step short of the design that dominates modern language implementations, and it said so: the VM is built on exactly the substrate a just-in-time compiler starts from. A JIT is the resolution of a dilemma the book has circled several times. Interpreting bytecode is cheap to *start* — no compilation, just dispatch — but pays a dispatch cost on every instruction forever. Compiling to native code is the reverse — expensive to start, because you must run the whole back end, but then free of dispatch on every subsequent execution. The interpreter wins on code that runs once; the compiler wins on code that runs a million times. A JIT refuses to choose in advance and instead *measures*.

Figure 15.5 shows the mechanism, and it is gratifyingly modest given how much it buys. The VM gains a counter on each loop header and function entry. Most code never trips it; that code stays interpreted, and pays nothing for the JIT's existence. The moment

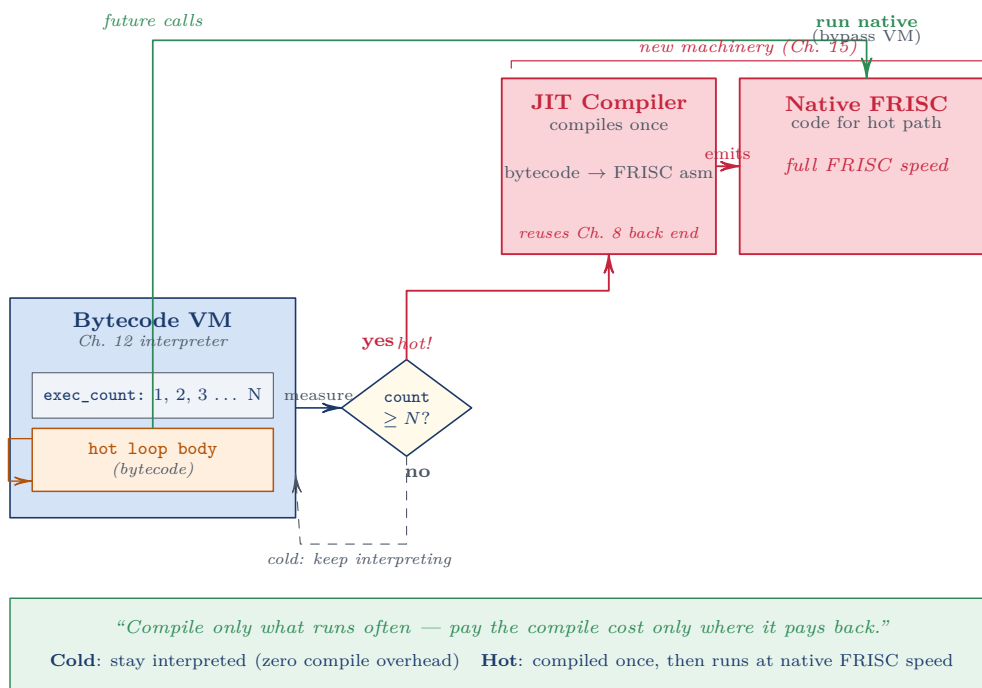


Figure 15.5: A JIT added to the bytecode VM of Chapter 12. The VM interprets bytecode as before, but now counts how often each loop or function runs. Cold code stays interpreted — cheap to start, never worth compiling. When a counter crosses a threshold the code is *hot*, and the JIT compiles that bytecode to native FRISC — reusing the code generator of Chapter 8 — so every later execution of the hot path runs native and bypasses the dispatch loop entirely. Pay the compile cost only where it pays back.

a counter crosses its threshold, the run-time system has learned something the compiler could not have known statically — that this particular path is hot — and it acts on the knowledge: it hands that path’s bytecode to a compiler that emits native FRISC, installs the result, and redirects future executions to the native version. The hot loop, the three per cent Knuth told us to care about in [Chapter 1](#), now runs at full machine speed; the cold ninety-seven per cent was never compiled and never needed to be. The defining move is that the decision is made at run time, on evidence, by the running program about itself.

The most striking thing about building this on FRISCcc is how much of it already exists. The compiler that the JIT invokes on the hot path is the back end of [Chapter 8](#) — IR or bytecode in, FRISC out — which we have already written, tested, and proved things about. The hard new parts are not in the compiler at all. They are in the *plumbing*: the profiling counters and their thresholds; deciding the unit of compilation (a whole method, as early Java JITs did, or a hot *trace* through whatever methods it crosses, as the tracing JITs pioneered for dynamic languages did [30]); patching a running program so that the next call to a function reaches its freshly compiled version without disturbing the call already in progress; and deciding when a compiled assumption has been invalidated and the code must be thrown away and re-interpreted. None of that is conceptually deep, but all of it is the kind of careful systems engineering that does not appear in a chapter on parsing. Aycock’s short history is the gentlest way in [8], and Smith and Nair’s book is the systematic treatment [76].

Key insight

A JIT is mostly a back end you already have, plus a thermometer. The compiler a just-in-time system runs on the hot path is an ordinary code generator — ours, from [Chapter 8](#). What makes it a JIT is everything *around* the compiler: the counters that decide what is hot, the machinery that installs native code into a running program, and the policy for when to give up on a compiled assumption. Having built the back end and the VM, you are most of the way to a JIT; the remaining distance is systems work, not compiler theory.

15.6 A different target, a different source

The last open question is the one [Chapter 13](#) kept running into and could only report: some costs do not live in the program. GCD spent three-quarters of its instructions inside the software F_MOD helper; CRC, more than four-fifths inside the same routine; the fixed-point loops, the majority inside F_FMUL. No pass of the optimiser touches those cycles, because they are not redundancy the compiler introduced — they are the price of asking a reduced instruction set to do arithmetic it has no instruction for. The optimiser, [Chapter 14](#) put it, cannot remove work the ISA imposes.

So change the ISA. The cheapest performance win available to any of these programs is not a cleverer compiler at all; it is a FRISC with a hardware multiplier and divider. A

hardware MUL costing a handful of cycles in place of a software F_MUL costing dozens would erase three-quarters of GCD's cost at a stroke, and would do it without changing one line of FRISCcc above the code generator — the lowering of a mul instruction would simply emit a hardware MUL instead of a CALL F_MUL, and the helper would fall away unused. This is the trade the original RISC argument was always explicit about: a smaller instruction set is cheaper to build in silicon and charges for the omission in software, every time a program needs the operation that was left out [35, 68]. Reading a program's helper-cycle budget is, in effect, reading a shopping list for the hardware designer.

Retargeting goes further: not a richer FRISC but a different machine entirely — ARM, RISC-V, x86. Here the $M + N$ promise of Chapter 1 finally comes due. Everything from the lexer through the optimiser produces and consumes target-independent IR and would not change at all. What you would write is a new code generator — a new instruction selector, a new register allocator tuned to the new register file, a new realisation of the calling convention — and a new set of runtime helpers, or none, if the new machine has the arithmetic in hardware. The simulator of Chapter 10 would be replaced by the real machine or its emulator. The work is real, but it is bounded and it is *local*: a back end is a self-contained project precisely because the IR drew a clean line for it to start from.

And the same line works in the other direction. A new *front end* — a different source language lowered to the same IR — inherits the entire back end for free. Point a Pascal parser, or a small functional language, or your own invented syntax at FRISCcc's IR, and the optimiser, the code generator, the interpreter, and the VM all work unchanged, because none of them ever knew or cared that the IR came from C. This is the deepest reason the book spent a whole chapter justifying the IR's design one decision at a time. An intermediate representation is not plumbing between two phases; it is the contract that makes every phase on either side of it independently replaceable. Build a second front end and you have proved the front half of the hourglass; build a second back end and you have proved the back half; do both and you have a compiler construction kit, which is what GCC and LLVM [53] in fact are.

Key insight

The IR is the unit of reuse. A hardware multiplier changes the back end and nothing else. A new target machine is a new back end and nothing else. A new source language is a new front end and nothing else. Every one of these large-sounding projects is bounded to one side of the IR, and that boundedness is not luck — it is the return on the design discipline of Chapter 6, collected in full only now, at the end, when there is finally a whole machine to extend.

Practice

These are not exercises with answers in Appendix F. They are the smallest honest versions of the projects this chapter sketched — each one a weekend, not an afternoon, and each

one a real change to the compiler you now understand well enough to make.

Exercise 15.1 (Conceptual · ★☆☆)

Trace the reuse boundary. Pick one extension from [Figure 15.2](#) — a hardware multiplier, a register VM, a new target, a new front end — and write down, phase by phase, exactly which phases of the pipeline ([Figure 15.1](#)) you would have to modify and which you would leave untouched. Justify each “untouched” against the artifact that phase produces or consumes. The shorter your list of modified phases, the better the IR boundary is doing its job.

Exercise 15.2 (Applied · ★★☆☆)

Lower one IR line to a register VM. Take a basic block of three or four instructions from any optimised IR listing in [Chapter 13](#). Hand-lower it twice: once to the stack bytecode of [Chapter 12](#), and once to the register bytecode sketched in [Figure 15.3](#). Count the dispatches each version costs. Does the reduction you find match the substantial drop Shi and colleagues report [74]? Explain any discrepancy in terms of how many of your operands were reused.

Exercise 15.3 (Conceptual · ★☆☆)

Convert a diamond to SSA by hand. Write a tiny program with an if/else that assigns the same variable in both branches and reads it afterward. Draw its control-flow graph, rename the assignments into SSA, and place the phi-node, as in [Figure 15.4](#). Then argue, for one specific use in your program, why copy propagation is a single unconditional rewrite under SSA but required a reaching-definitions check on the plain IR of [Chapter 7](#).

Exercise 15.4 (Applied · ★★☆☆)

Find the JIT threshold. Using the executed-instruction counts of [Chapter 14](#), estimate for one looping program how many iterations it would have to run before compiling its hot loop to native FRISC paid for the cost of compiling it. State your assumption about the per-iteration saving and the one-time compile cost. What does your answer say about which of the fourteen suite programs a JIT would bother to compile at all?

Exercise 15.5 (Conceptual · ★☆☆)

The honest aggregate, one last time. Chapter 14 reported a suite-wide helper share of 5.7% that concealed a per-program spread from zero to 85.7%. Suppose the FRISC gained a hardware multiplier and divider. Predict qualitatively what happens to each end of that spread, and explain why the aggregate is once again the wrong number to quote to someone deciding whether the hardware change is worth it.

15.7 Books and papers worth your time

A compiler book is a doorway, not a destination, and the field behind the door is enormous. What follows is short and opinionated — the handful of sources that would take you furthest from where this book leaves you, grouped by the road they open.

For the craft of compilation as a whole, the three references this book leaned on hardest remain the ones to own. Aho, Lam, Sethi, and Ullman's *Compilers* [2] — the Dragon Book — is still the canonical account of lexing, LR parsing, and dataflow, and it is the vocabulary the rest of the field speaks. Cooper and Torczon's *Engineering a Compiler* [18] is the more modern and more practical companion, especially strong on the back end and on the pass-ordering questions Chapter 14 measured. Muchnick's *Advanced Compiler Design and Implementation* [62] is the encyclopaedia of optimisations when you need the one this book did not build. And Appel's *Modern Compiler Implementation* [7] is the book to read next if you want to write the whole thing again from a different angle, with SSA and register allocation at the centre rather than the margin.

For the roads this chapter sketched, go to the sources. SSA is Cytron and colleagues [21]: read it for the dominance-frontier algorithm and the phi-placement argument, which is the one genuinely clever idea in the whole representation, and Lengauer and Tarjan [54] for the fast dominator computation it rests on. Graph-colouring register allocation begins with Chaitin [15] and reaches its practical form in George and Appel's iterated coalescing [31]; together they are the back end's answer to the stack-slot inefficiency we papered over with load forwarding. The stack-versus-register VM question is Shi and colleagues [74], and the interpreter-dispatch costs that make it matter are Ertl and Gregg's [26]. And for just-in-time compilation, Aycock's history [8] is the overview, Smith and Nair [76] the systematic treatment, and Gal and colleagues [30] the paper that made tracing JITs concrete.

For the foundations underneath all of it — what the architecture trade in Section 15.6 really costs, why a reduced instruction set charges in software for what it omits in silicon — Hennessy and Patterson's *Computer Architecture* [35] is the reference, and Patterson's original RISC argument [68] is still worth reading in the original. And for the book that showed this one how to be written — warm, first-person, build-along, and rigorous all at once — Nystrom's *Crafting Interpreters* [67] stops at a bytecode VM, which is exactly where Chapter 12 of this book arrives, and the two make a natural pair: he ends where we end Part IV, and this book carries on through the native back end he chose not to build.

That is the reading list. The compiler is on the bench, the roads out are signposted, and there is one thing left to say.

One last word

We began with a program that computed five times three and a black box that turned it into the answer. We end with the box dismantled on the bench, every part of it named and understood, and a map of the parts we chose not to build. Between those two points is the whole of a compiler: a lexer that recognises words, a parser that finds structure, a type checker that insists on meaning, an intermediate representation that decouples the source from the machine, an optimiser that earns its keep unevenly and honestly, a code generator that makes every abstraction concrete, a runtime that fills the hardware's gaps, a simulator that finally runs the thing, and — because the IR made it cheap — two more ways to run it that share nothing but their agreement on the answer.

The thing worth carrying away is not any one of those phases. It is the shape they make together: a pipeline of small, individually understandable, individually checkable transformations, each consuming one well-defined artifact and producing the next, decoupled at a representation careful enough that either half can be replaced without disturbing the other. That shape is not specific to C, or to FRISC, or to FRISCcc. It is how compilers are built, and now that you have seen one built in full, you have seen them all in outline — and, more to the point, you can build your own.

So build it. Add the struct, or the register allocator, or the SSA pass, or the second target. Point a new front end at the IR and watch the back end you understand swallow a language it has never seen. Put a counter on a loop and compile the hot path. The machine on the bench is yours to take apart further, and the best way to find out what compilers can do is the one Alan Kay named in the epigraph: invent the future you want by building it. The black box is open. Go and open the next one.

Appendix A

A Note on Sources

A.1 Lineage

FRISCcc and the book around it descend from the course *Prevodenje programskih jezika* (PPJ; in English, Programming Language Translation, PLT) taught at the Faculty of Electrical Engineering and Computing, University of Zagreb. The course’s grammar, terminal-token naming convention, and the FRISC instruction set come from that material [9, 27], and the standard Croatian-language text for the course is Srbljić’s *Prevodenje programskih jezika* [77], with its companion volume *Uvod u teoriju računarstva* [78] covering the automata and formal-language theory the front end rests on. What is new here is the engineering: the implementation language (Java 21), the build system, the IR design, and all of the prose are this book’s own. The complete compiler is available as open source [47], and every trace, listing, and benchmark in these pages is regenerated from it. Where the underlying source material is in Croatian, we translate the engineering content into English and preserve the Croatian identifiers where they appear in files we will actually open.

A.2 Croatian–English non-terminal mapping

The grammar file `config/parser_definition.txt` names every non-terminal in Croatian, because that is the language of the original PLT-FER course material. Chapter prose uses the English translation everywhere; this table is the bridge. Terminal-token names (KR_INT, IDN, L_ZAGRADA, and so on) follow a prefix convention rather than appearing in the table: the prefix KR_ marks a keyword, L_ a left bracket of some kind, D_ a right bracket, and OP_ an operator. The grammar file also declares a handful of non-terminals for `struct` and pointer constructs that fall outside the C subset this book actually compiles; the table below covers only the non-terminals reachable in the supported subset, which is every non-terminal we will meet in these pages.

Croatian non-terminal	English
prijevodna_jedinica	translation unit
vanjska_deklaracija	external declaration
definicija_funkcije	function definition
slozena_naredba	compound statement
lista_naredbi	list of statements
naredba	statement
izraz_naredba	expression statement
naredba_grananja	branching statement
naredba_petlje	loop statement
naredba_skoka	jump statement
izraz	expression
izraz_pridruzivanja	assignment expression
log_ili_izraz	logical-or expression
log_i_izraz	logical-and expression
bin_ili_izraz	bitwise-or expression
bin_xili_izraz	bitwise-xor expression
bin_i_izraz	bitwise-and expression
jednakosni_izraz	equality expression
odnosni_izraz	relational expression
aditivni_izraz	additive expression
multiplikativni_izraz	multiplicative expression
cast_izraz	cast expression
unarni_izraz	unary expression
postfiks_izraz	postfix expression
primarni_izraz	primary expression
lista_argumenata	argument list
inicijalizator	initialiser
lista_deklaracija	list of declarations
deklaracija	declaration
lista_init_deklaratora	list of init-declarators
init_deklarator	init-declarator
specifikator_tipa	type specifier
deklarator	declarator
izravni_deklarator	direct declarator
lista_parametara	parameter list
deklaracija_parametra	parameter declaration

A.3 Reading the original materials

If you want to trace the compiler's lineage back to the original course material, there are two public starting points. The course itself, *Prevođenje programskih jezika*, publishes its syllabus, lecture notes, and laboratory assignments through the University of Zagreb's Faculty of Electrical Engineering and Computing; the public course page [27] is the canonical entry point and is kept current from one academic year to the next. The FRISC processor and its assembler are documented separately in Basch and Kovač's *Osnove procesora FRISC* [9]

and in the processor description that accompanies the simulator we use throughout the book ([Appendix C](#)); the simulator’s own repository [88] carries an instruction summary and worked assembly examples as well.

The original PLT-FER material is in Croatian, and a reader who does not read Croatian can still extract most of its value. Every formal construction the course presents — Thompson’s NFA assembly, the subset construction, the canonical collection of LR(1) items, the shift/reduce table build — uses notation that is, by design, language-independent, and it is the same notation this book uses. Where a chapter leans on a specific construction from the course, the Notes and Further Reading section at the end of that chapter cites the course material [27] alongside the standard English-language references — Aho and colleagues [2], Grune and Jacobs [33], and the primary sources — so a reader can triangulate any construction against a text in a language they read. A reader who does read Croatian will find the fullest treatment in Srbljić’s two volumes: *Prevodenje programskih jezika* [77] for the compiler pipeline end to end, and *Uvod u teoriju računarstva* [78] for the regular languages, finite automata, and context-free grammars that [Chapter 3](#) and [Chapter 4](#) build on. Nothing in the body of this book requires reading the Croatian originals; they are offered as provenance, not as a prerequisite.

Appendix B

FRISC ISA Reference

The chapters describe FRISC only as far as the code generator needs it; this appendix gives the whole instruction set — every mnemonic, every operand field, every flag — together with the register model, the encoding, the memory layout, and the assembler syntax in which the compiler’s output is written. Nothing here is FRISCcc-specific except the calling convention and the runtime helper set, collected in the last two sections; the rest is the processor as the simulator of [Appendix C](#) implements it [9, 88].

B.1 The processor at a glance

FRISC — the FER Reduced Instruction-Set Computer — is a 32-bit load/store RISC processor. “Load/store” is the single most important fact about it: arithmetic and logic operate only on registers and immediates, never directly on memory, so the only instructions that touch memory are the explicit loads and stores. Every instruction is exactly one 32-bit word, four bytes, fixed length, which is what makes the fetch step trivial — the program counter always advances by four — and what makes the decode step a matter of slicing fixed bit-fields out of a known-width word.

The processor has eight general-purpose 32-bit registers, a program counter, and a status register holding the condition flags. It executes the textbook cycle: fetch the word at the program counter, decode its fields, execute its effect, and repeat until a HALT ([Figure B.1](#)). There is no pipeline, no cache, and no memory-management unit to model; one instruction completes before the next begins, which is exactly why the executed-instruction counts the book reports are deterministic and reproducible.

B.2 Register file

The eight general-purpose registers are named `R0` through `R7`. The hardware treats them identically; the distinctions that matter to a compiler are entirely a matter of convention, fixed by the calling convention in [Section B.9](#) and summarized in [Figure B.2](#). Two more registers are visible to the programmer but addressed specially: the program counter `PC`, which holds the address of the next instruction to fetch, and the status register `SR`, which

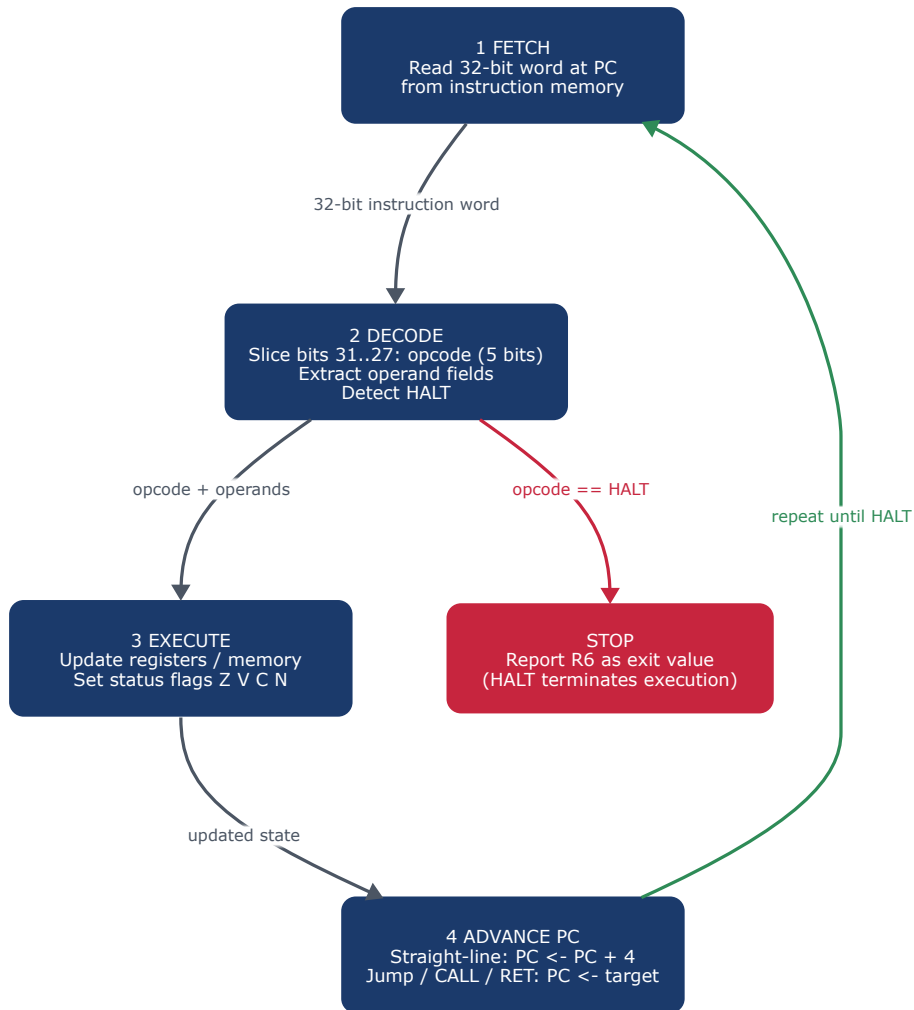


Figure B.1: The fetch–decode–execute cycle the simulator runs once per `performCycle`. The program counter selects a 32-bit word; the decoder slices its opcode and operand fields; the execute step updates registers, memory, and the status flags, and advances the program counter — by four for straight-line instructions, or to a target for taken jumps. HALT is the only instruction that breaks the loop.

holds the condition flags and the interrupt-control bits (Section B.5). PC is not a general-purpose register — you change it by jumping, not by naming it as an ALU destination — and SR is reachable only as the special source or destination of a MOVE.

General-Purpose Registers	
<i>(all 32-bit; roles are software convention, not hardware constraint)</i>	
R0	primary expression result (caller-saved)
R1	scratch / argument (caller-saved)
R2	scratch / argument (caller-saved)
R3	scratch / argument (caller-saved)
R4	scratch / argument (caller-saved)
R5	frame pointer (FP) (callee-saved)
R6	return-value register (callee-saved)
R7	stack pointer (SP) — init 40000, grows ↓ (callee-saved)
— special registers —	
Special Registers	
PC	program counter — 32-bit byte address of the next instruction <i>(not directly writeable; modified by branches and calls)</i>
SR	status register — condition flags + interrupt control low bits: Z (zero) V (overflow) C (carry) N (negative)
Note: R0–R7 are architecturally identical to the hardware. The roles above are the FRISCcc calling convention; hand-written code may differ.	

Figure B.2: The FRISC register file. All eight general registers are interchangeable to the hardware; the roles shown — result, scratch, frame pointer, return value, stack pointer — are the FRISCcc calling convention of Section B.9, not a hardware property. PC and SR are special registers, reached by jumping and by MOVE respectively.

Every register is 32 bits wide and every register holds an uninterpreted bit pattern. Whether a given word denotes a signed integer, an unsigned integer, a pointer, or a fixed-point fraction is decided entirely by the instruction that reads it: ADD treats its operands as

+524,287; a constant outside that range cannot ride inside a single instruction and must instead be assembled into memory and loaded, or built up from smaller pieces. The same 20-bit field doubles as a memory address in the direct-addressing memory instructions and as a displacement in the indirect-with-offset form, which is why a single instruction can name an absolute address only within the low megabyte; larger addresses are formed in a register and used indirectly.

MOVE reuses the register fields but adds two special cases: bit 21 set means the *source* is the status register SR, and bit 20 set means the *destination* is SR. These are how a program saves and restores the flags. The jump and call instructions replace the ALU layout with a four-bit *condition* field in bits 25–22 (Section B.5) and a target that is either a register (bits 19–17) or a 20-bit immediate (bits 19–0), again selected by the mode bit 26.

B.4 Instruction reference, by mnemonic

The complete opcode map is given in Table B.1: twenty-seven instructions, each a five-bit opcode. We then describe each class in turn. Throughout, Rd is the destination register, Rs1 and Rs2 the source registers, imm a 20-bit sign-extended immediate, and <src2> the second operand, which is either Rs2 or imm according to the mode bit. The phrase “sets flags” means the instruction updates Z, V, C, and N as described in Section B.5.

Opcode	Mnemonic	Opcode	Mnemonic
00000	MOVE	10000	POP
00001	OR	10001	PUSH
00010	AND	10010	LOADB
00011	XOR	10011	STOREB
00100	ADD	10100	LOADH
00101	ADC	10101	STOREH
00110	SUB	10110	LOAD
00111	SBC	10111	STORE
01000	ROTL	11000	JP
01001	ROTR	11001	CALL
01010	SHL	11010	JR
01011	SHR	11011	RET
01100	ASHR	11111	HALT
01101	CMP		

Table B.1: The twenty-seven FRISC opcodes, each a five-bit field in instruction bits 31–27. Opcodes 01110 and 01111 (between the ALU and memory groups) are unused, as are 11100–11110. RET encodes its two interrupt-return variants, RETI and RETN, in its low bits.

B.4.1 Data movement

MOVE copies a word without arithmetic. In its register form, `MOVE Rs, Rd` copies `Rs` to `Rd`; in its immediate form, `MOVE imm, Rd` loads a sign-extended 20-bit constant. The two special forms `MOVE SR, Rd` and `MOVE Rs, SR` read and write the status register, and a program initializing the stack begins with `MOVE 40000, R7`. MOVE does not set flags except when its destination is `SR`, in which case it sets all of them at once by definition.

B.4.2 Arithmetic and logic

The arithmetic instructions take the three-operand ALU form `OP Rs1, <src2>, Rd`, computing $Rd \leftarrow Rs1 \text{ op } \langle src2 \rangle$ and setting flags. `ADD` and `SUB` are the ordinary two's complement add and subtract. `ADC` (add with carry) and `SBC` (subtract with carry/borrow) fold the `C` flag into the operation, which is how multi-word arithmetic is chained — `ADC` is what the runtime's fixed-point multiply helper (Section B.8) uses to maintain its 64-bit intermediate product. `SBC` is provided by the ISA but unused by the generated code. `AND`, `OR`, and `XOR` are the bitwise Boolean operations and set `Z` and `N` from their result while clearing `C` and `V`.

`CMP Rs1, <src2>` is a `SUB` that discards its difference and keeps only the flags. It is the instruction that precedes every conditional jump: the comparison sets `Z`, `V`, `C`, and `N`, and the following `JP` reads them through its condition field. Because `CMP` sets both the carry-based unsigned flags and the sign/overflow-based signed flags, one comparison serves both a signed and an unsigned branch — the branch, not the compare, chooses the interpretation.

B.4.3 Shifts and rotates

`SHL` and `SHR` are the logical left and right shifts, filling with zeros; `ASHR` is the arithmetic right shift, replicating the sign bit, which is what distinguishes a signed division-by-power-of-two from an unsigned one. `ROTL` and `ROTR` rotate, recirculating the bits that fall off one end into the other. All five take the ALU form, shifting `Rs1` by the amount in `<src2>` into `Rd`, and all five set flags; the last bit shifted out lands in `C`.

B.4.4 Memory access

The load/store instructions are the only ones that read or write memory. `LOAD Rd, (addr)` reads a 32-bit word; `STORE Rs, (addr)` writes one. The half-word and byte variants — `LOADH/STOREH` for 16 bits and `LOADB/STOREB` for 8 — move narrower quantities, zero-extending on load. The address is either direct, written `(label)` and carried as a 20-bit field, or register indirect with an optional displacement, written `(Rb+offset)`; the displacement is the same 20-bit field. `FRISCcc` emits the indirect form almost exclusively, because every local lives at a fixed offset from the frame pointer `R5` (Section B.9). None of the memory instructions sets flags.

PUSH *Rs* and POP *Rd* are the stack instructions and the only memory instructions with an implied address. PUSH first decrements *R7* by four and then stores *Rs* at the new top; POP reads the word at *R7* and then increments *R7* by four. The stack therefore grows toward lower addresses, which is the fact behind the frame layout of [Chapter 9](#) and behind the stack-overflow arithmetic in the exercise of [Chapter 2](#).

B.4.5 Control flow

JP jumps, CALL calls, RET returns, and JR jumps relative; all four carry the four-bit condition field, so every one of them is conditional. JP *<target>* sets the program counter to its target, a register or a 20-bit immediate address. CALL *<target>* first pushes the return address — the address of the instruction after the call — onto the stack exactly as PUSH would, then jumps; RET pops that address back into the program counter. JR adds a signed displacement to the program counter rather than naming an absolute target, which keeps a short forward or backward branch independent of where the code is loaded. Written with a condition suffix — JP_EQ, CALL_NZ, RET_C, and so on — the instruction takes effect only when the flags satisfy the condition; written bare, it is unconditional. RET additionally encodes the two interrupt-return forms RETI and RETN in its low bits, which re-enable interrupts on the way out; FRISCCc's generated code uses only the plain RET. None of the control instructions sets flags.

HALT stops the processor. It is the one instruction that ends the fetch–decode–execute loop rather than advancing to a successor, and it is what the simulator waits for before reporting the result ([Appendix C](#)). HALT, too, carries the four-bit condition field and so admits a conditional form, though FRISCCc emits only the unconditional one. FRISCCc places a single HALT in the bootstrap prologue, reached when main returns, never at the lexical end of the program text.

B.5 Condition codes and flags

The status register holds four condition flags, set by the arithmetic, logic, shift, and compare instructions and read by the conditional jumps. Z is set when the result is zero. N is set when the result's top bit is one, that is, when it is negative read as a signed two's-complement value. C is the carry out of the top bit, which on a subtract or compare doubles as the unsigned borrow. V is signed overflow: it is set when the result's sign is not the one two's-complement arithmetic should have produced. [Table B.2](#) lists their bit positions in the status register, alongside the interrupt-control bits that share the register but play no part in compiled code.

A conditional jump names one of fifteen conditions in its four-bit condition field; [Table B.3](#) gives the full set, with the flag expression each one tests. The signed and unsigned orderings are distinct conditions over the same flags: _SLT (signed less than) tests whether N and V differ, while _ULT, written _C, tests the carry directly. This is why the code generator must

Bit value	Name	Meaning
1	N	negative (result's top bit set)
2	C	carry / unsigned borrow
4	V	signed overflow
8	Z	zero result
16	EINT0	enable interrupt level 0
32	EINT1	enable interrupt level 1
64	EINT2	enable interrupt level 2
128	GIE	global interrupt enable
256	INT0	interrupt 0 pending
512	INT1	interrupt 1 pending
1024	INT2	interrupt 2 pending

Table B.2: The status-register layout. The four condition flags occupy the low bits; the interrupt-control bits above them are present in the hardware but unused by compiled FRISCcc programs, which neither enable interrupts nor service them.

know the signedness of a comparison even though `CMP` does not: the distinction lives in the branch, in the choice of condition suffix.

Field	Suffix	Tested condition
0000	(none)	always
0001	_N / _M	N set (minus)
0010	_NN / _P	N clear (plus)
0011	_C / _ULT	C set (unsigned less than)
0100	_NC / _UGE	C clear (unsigned $\geq$)
0101	_V	V set (overflow)
0110	_NV	V clear
0111	_Z / _EQ	Z set (equal)
1000	_NZ / _NE	Z clear (not equal)
1001	_ULE	unsigned less or equal
1010	_UGT	unsigned greater than
1011	_SLT	signed less than
1100	_SLE	signed less or equal
1101	_SGE	signed greater or equal
1110	_SGT	signed greater than

Table B.3: The fifteen jump conditions, with the flag each tests where the table names one. Several have two spellings; the simulator accepts either. The unsigned orderings read the carry flag (and, for `_ULE`/`_UGT`, the zero flag); the signed orderings read N, V, and Z.

The lesson the code generator carries away from this table is that a single `CMP` feeds both interpretations, and the choice of which row to branch on is where the source expression's signedness finally lands in the emitted code.

B.6 Memory model

FRISC memory is a flat, byte-addressable array. A word is four bytes and is stored *little-endian*: the least significant byte sits at the lowest address, which is why the simulator's word writes lay down the low byte first and shift right for each higher byte. Word and half-word accesses are expected to be aligned — words on four-byte boundaries, half-words on two — and the code generator guarantees this by aligning every frame slot.

The address space the simulator presents is configurable; FRISCcc configures it with a 1000 KiB space — larger than the friscjs default of 256 KiB, and comfortably larger than any example needs. [Figure B.4](#) sketches the conventional division of that space: program code loaded from address zero upward, static data above it, and the stack descending from its initialized top at address 40000. Because the stack grows down and the heap, when a program uses one, grows up from above the static data, the two approach each other from opposite ends of a shared region; the FRISC has no hardware guard between them, so a stack that grows too deep silently overwrites whatever lies below it. The compiled programs in this book do not use a heap, and their stacks are shallow, so the regions never meet in practice.

The hardware also supports memory-mapped input and output: an I/O unit can be wired to a range of addresses so that loads and stores to that range communicate with the device rather than with memory. The simulator implements this, and the FRISC course uses it for timers and simple peripherals, but compiled FRISCcc programs perform no I/O — their only observable output is the value returned from `main` ([Appendix C](#)) — so memory-mapped I/O does not appear in any generated code.

B.7 Assembler syntax and directives

The compiler's back end emits FRISC *assembly*, text that the simulator's assembler turns into the machine words just described. A line is an optional label, an optional instruction or directive, and an optional comment introduced by a semicolon. A label is a name in the first column; it stands for the address of the line it marks and is how jumps and direct loads refer to code and data without writing numeric addresses.

Six pseudo-operations, written with a leading backtick, direct the assembler rather than emitting an instruction. ``ORG` sets the assembly location counter, placing what follows at a chosen address. ``DW` (define word), together with the bare forms `DW`, `DH`, and `DB` for word, half-word, and byte, deposits constant data into memory — this is how a large constant that cannot fit a 20-bit immediate is materialized and then loaded. ``DS` (define space) reserves a block of uninitialized bytes. ``EQU` binds a name to a constant for readability. ``BASE` selects the default radix in which subsequent numeric literals are read. ``END` marks the end of the source. A reader inspecting the compiler's `a.out` will see these directives framing the instruction stream; [Chapter 8](#) and [Chapter 9](#) walk through a generated file in detail.

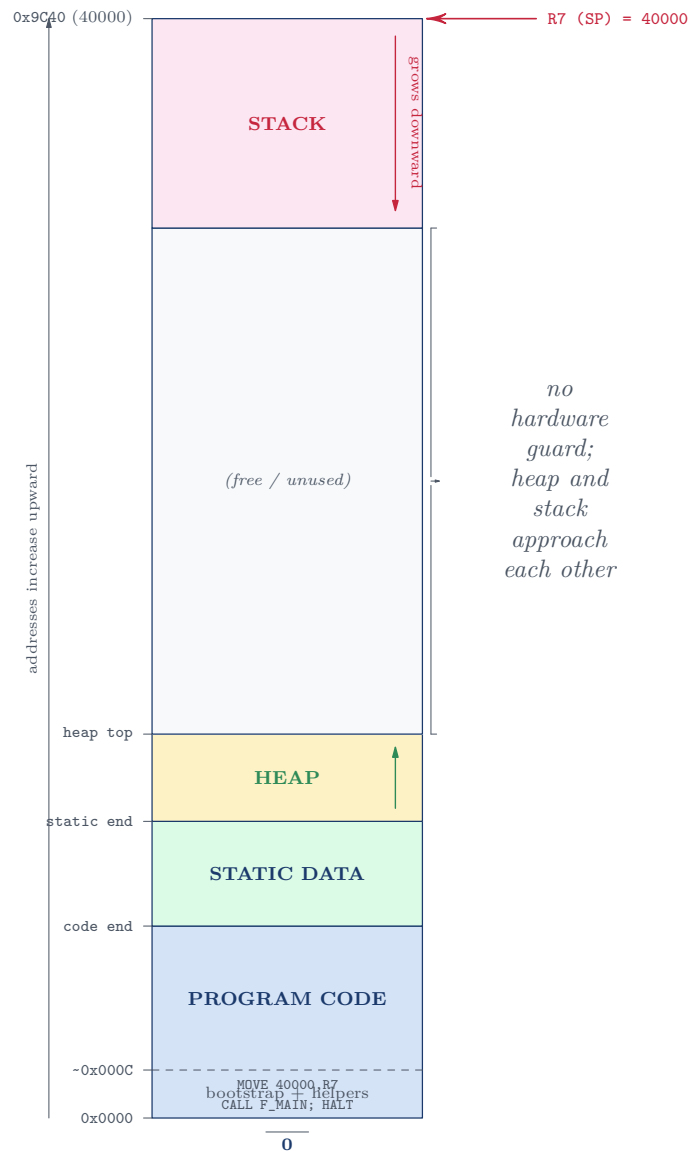


Figure B.4: The conventional layout of the FRISC address space as FRISCcc uses it. Code is loaded from address zero; static data sits above it; the stack is initialized to descend from address `40000`. Addresses increase upward in the figure; the stack pointer `R7` moves downward as frames are pushed. There is no hardware boundary between the stack and the data below it.

B.8 The runtime helper set

FRISC has no hardware multiply, no divide, and no floating-point unit. Operations the source language treats as primitive therefore compile not to single instructions but to calls into a small library of *runtime helpers* — hand-written FRISC routines that the code generator emits once per program and calls wherever the operation is needed. Table B.4 lists them. Each follows the same lightweight convention: arguments arrive in the scratch registers, the result comes back in R6 like any return value, and the helper preserves whatever else it touches.

Helper	Operation
F_MUL	32-bit integer multiply
F_DIV	32-bit signed integer division
F_MOD	32-bit signed integer remainder
F_FML	fixed-point (Q16.16) multiply
F_FDIV	fixed-point (Q16.16) division
F_I2F	integer to fixed-point conversion
F_F2I	fixed-point to integer conversion
L_BOUNDS_ERROR	array-bounds-violation trap

Table B.4: The runtime helpers the code generator emits to stand in for operations the hardware lacks. Integer multiply, divide, and remainder are built from ADD, SUB, and the shifts; the floating-point operations use a Q16.16 fixed-point representation; L_BOUNDS_ERROR is the trap a failed bounds check jumps to. Unlike the others, it is a code-emitted label reached by a conditional jump, not an F_-prefixed routine that is CALLED.

The integer multiply, divide, and remainder helpers implement the usual shift-and-add and shift-and-subtract restoring-division algorithms out of ADD, SUB, and the shifts. ADC earns its keep elsewhere: the fixed-point multiply F_FML uses it to maintain the 64-bit intermediate product before the 16-bit right shift. The ISA provides SBC, but the generated code never emits it. The floating-point helpers do not use IEEE 754; the source language's float is represented in *Q16.16 fixed point*, a 32-bit word split into a 16-bit integer part and a 16-bit fraction, so that F_FML is an integer multiply followed by a 16-bit right shift and F_FDIV a left shift followed by a divide. F_I2F and F_F2I convert between the integer and fixed-point representations by shifting. L_BOUNDS_ERROR is the destination of every array-bounds check: when an index falls outside its array, control jumps here, and the trap halts the machine rather than letting a corrupt access proceed. Chapter 9 derives each helper from its algorithm and shows the generated calls in context; the helpers' share of executed instructions, which dominates the cost of some programs, is measured in Chapter 14.

Key insight

A FRISC float is a scaled integer, not IEEE 754. The source language's float is stored in Q16.16 fixed point: a 32-bit word whose high 16 bits are the integer part and whose low 16 bits are the fraction. The stored word is therefore the real value times $2^{16} = 65536$. To read a float result the simulator reports (Section C.3), divide the printed integer by 65536; the word 360448, for instance, denotes $360448/65536 = 5.5$.

B.9 The calling convention

The hardware fixes none of the register roles; the compiler does, and the agreement it imposes is what lets a caller and a callee that were generated independently nonetheless fit together. R0 carries the primary result of an expression and is caller-saved. R1 through R4 are scratch and argument registers, also caller-saved. R5 is the frame pointer: every local variable and parameter lives at a fixed displacement from it, which is why the generated code reaches locals through the register-indirect-with-offset memory form. R6 is the return-value register — the value a function returns, and the value the simulator reports when the program halts. R7 is the stack pointer, initialized to 40000 and moved downward by PUSH, CALL, and frame setup.

A call therefore proceeds as follows. The caller places arguments in the scratch registers, saves whatever caller-saved registers it still needs, and issues CALL, which pushes the return address. The callee establishes its frame by saving the old frame pointer and setting R5 to the new frame, reserves space for its locals by lowering R7, runs its body reaching locals at offsets from R5, leaves its result in R6, restores R5 and R7, and issues RET, which pops the return address back into the program counter. The bootstrap that the code generator emits ahead of the user's code — `MOVE 40000, R7; CALL F_MAIN; HALT` — is the base case of this protocol: it initializes the stack, calls `main`, and halts with `main`'s result sitting in R6. Chapter 9 follows a complete call through frame setup and teardown.

Appendix C

Simulator Reference

The back end of [Chapter 8](#) emits FRISC assembly; turning that assembly into machine words and executing them is the job of a separate tool, the FRISCjs simulator [\[88\]](#). FRISCcc does not run its own output — it hands that task to the simulator. This appendix describes the tool: where to get it, the two interfaces it offers, the library the compiler drives under the hood, and — the part we reach for day to day — how to read what comes out. The instruction set the simulator implements is the subject of [Appendix B](#); here we treat it as a black box that runs FRISC and reports a result.

C.1 Where to find it

FRISCjs is an open-source FRISC assembler and processor simulator written in JavaScript [\[88\]](#). It is two cooperating pieces: an *assembler* that parses FRISC assembly text and produces machine code, and a *CPU simulator* that loads that machine code into a memory image and executes it one instruction at a time. The project’s documentation — an API reference and worked examples — ships with the source, and the FRISC processor it models is described in the textbook *Osnove procesora FRISC* [\[9\]](#). A reader who wants to single-step a program by hand, set breakpoints, and watch the registers change will find the project’s browser-based interface the most comfortable way in; it presents an editor, a register panel, and a memory view, and it runs entirely client-side. For batch execution — compile a program, run it, read one number — the command-line path that FRISCcc itself uses is faster, and we describe it next.

C.2 How FRISCcc drives the simulator

When we ask the compiler to run a program rather than merely to compile it ([Appendix E](#)), FRISCcc shells out to the simulator’s library and executes the assembly it just produced. The mechanism is worth understanding, because it explains both what the result means and why the cycle counts the book reports are exact.

The compiler hands the assembler the generated assembly as a single string; the assembler returns a memory image, an array of byte values ready to load. The compiler then constructs a simulator instance, sizes its memory, loads the image, and runs the processor by calling

its single-step primitive in a loop: fetch, decode, execute, advance, repeat. The loop is not driven by a wall-clock timer — it is a plain counted loop — so each iteration is exactly one instruction, and counting iterations counts executed instructions precisely. The loop ends when the processor executes `HALT`, which the simulator signals by invoking its stop handler; at that point the compiler reads the result and prints it. A program that never reaches `HALT` would loop forever, so the driver also imposes a generous step ceiling — two hundred million instructions by default — and aborts with a diagnostic if the ceiling is crossed, which catches an accidental infinite loop without hanging the build.

The memory image the simulator runs in is larger than any example needs: the compiler sizes it at roughly a megabyte, far above the few kilobytes a compiled program touches. The stack lives near the top of that image, descending from address `40000` (Section B.6); the code and its constants occupy the bottom.

C.3 Reading the output

The single number a run prints is the contents of register `R6` at the moment the processor halts. By the calling convention (Section B.9), `R6` is the return-value register, and the bootstrap the compiler emits — initialize the stack, call `main`, halt — arranges that when `main` returns, its value is sitting in `R6` when `HALT` runs. So the printed number *is* the value the program's `main` returned, interpreted as a two's-complement 32-bit integer — a negative return such as `-1` prints with its sign, not as an unsigned magnitude. There is no other output: the compiled programs in this book perform no input and no output of their own, so a program communicates its result solely through its return value, exactly as the small-step semantics of Appendix G formalizes.

A float-returning program prints the integer *bit pattern* of its Q16.16 fixed-point result, not a decimal fraction, because `R6` holds 32 raw bits and the simulator reports them as an integer. To read such a result as a number, divide it by $2^{16} = 65536$: the value `360448`, for instance, is $360448/65536 = 5.5$. The fixed-point representation is described in Section B.8.

For finer-grained observation than a single final number, the simulator exposes the processor state directly. The register file — `R0` through `R7`, the program counter, and the status register — is readable at any point, and the simulator offers hooks that fire before and after each cycle, which is how an external harness can trace a run instruction by instruction or tally how many instructions fall in each routine. The book's measured instruction counts (Chapter 14) and per-routine helper budgets come from exactly this mechanism: a counting loop over single steps, with a per-cycle hook that buckets each executed instruction by the routine it belongs to. Because the simulator is deterministic and untimed, those counts are reproducible to the instruction — the same program always executes the same number of instructions, which is the property that makes the book's performance claims checkable rather than anecdotal.

C.4 Memory-mapped input and output

The FRISC hardware, and the simulator faithfully, support memory-mapped I/O: a peripheral can be attached to a range of addresses so that loads and stores to that range talk to the device instead of to memory, and some devices can raise interrupts. The FRISC course uses this facility to drive timers, counters, and simple displays, and the browser interface can attach such units. None of it appears in compiled FRISCcc programs. The source language has no I/O constructs, the code generator emits no device accesses, and the runtime enables no interrupts; a compiled program's entire interaction with the outside world is the value it leaves in R6. We mention memory-mapped I/O here only so that a reader exploring the simulator's full capabilities is not surprised to find a whole subsystem the compiler never exercises.

Appendix D

Language and IR Grammar

The compiler manipulates two languages: the C subset it accepts as input, and the typed intermediate representation it lowers that input into. This appendix is the formal reference for both. They are defined not by prose but by data — a handful of specification files the compiler reads at build time — and we describe those files first, because they are the single source of truth from which the lexer, the parser, the semantic analyzer, and the IR verifier are all driven. The English names used throughout the book for the grammar’s non-terminals correspond to Croatian identifiers in the specification files; the mapping is in [Section A.2](#).

D.1 The specification files

Four declarative specifications define the front end. Each is a plain text file in a small domain-specific format, consumed by the corresponding compiler phase. None contains executable code; each is a description from which a phase’s behaviour is generated or driven.

The *lexer specification* lists the named character classes, the scanner states, the complete token alphabet, and the rules that map matched text to tokens. The *parser specification* lists the grammar’s non-terminals, its terminals, the synchronization tokens used for error recovery, and the productions, from which the canonical LR(1) tables of [Chapter 4](#) are built. The *semantics specification* lists the productions over which the type rules of [Chapter 5](#) are defined — it is the grammar the semantic analyzer walks, annotated in the analyzer with the typing judgements. The *IR specification* gives the grammar of the textual intermediate representation, the form in which IR is printed, parsed back, and fed to the interpreter and the bytecode VM of Part IV. [Table D.1](#) names them and the phase each drives.

D.2 Lexical structure

The lexer specification opens with *named character classes* — abbreviations such as a class for letters, one for decimal digits, one for hexadecimal digits, one for whitespace, and one for “every character,” written as alternations and reused inside the rules so that no rule has to spell out a long alternation twice. It then declares the scanner’s *states* and the complete *token alphabet*, and finally the *rules*.

Specification	Drives	Defines
lexer	tokenizer (Chapter 3)	character classes, states, tokens, rules
parser	LR(1) parser (Chapter 4)	non-terminals, terminals, productions
semantics	type checker (Chapter 5)	the productions the type rules attach to
IR	IR printer/parser, Part IV back ends	the typed three-address IR grammar

Table D.1: The four declarative specifications and the phase each one drives. Each is plain text in a small format; together they fix the accepted language and the intermediate representation.

The scanner is a state machine with four states, listed in Table D.2. It begins in the start state; a `//` sends it into the line-comment state until the next newline, a `/*` into the block-comment state until the closing `*/`, and a double quote into the string state until the closing quote. Comments and the whitespace between tokens produce no token at all — the rule that matches them emits nothing — which is how the scanner discards them while still counting newlines for diagnostics.

Scanner state	Role
start	default; matches tokens, enters the other states
line comment	inside <code>// ...</code> ; discards to end of line
block comment	inside <code>/* ... */</code> ; discards, counts newlines
string	inside a string literal; accumulates until closing quote

Table D.2: The lexer’s four states. Each rule is guarded by the state it applies in; transitions are driven by the comment and string delimiters.

Each rule is a state guard, a regular expression, and a block of actions. The actions are few: emit a named token, emit nothing (for whitespace and comments), record a newline for line tracking, switch to another state, or roll the input position back so that a delimiter is reanalyzed in the state it switches into. Identifiers and numeric literals are matched by the longest-match (maximal-munch) rule of Chapter 3; a keyword such as `int` is recognized by a rule that precedes the identifier rule, so that a reserved word is never mistaken for a name.

The token alphabet is given in full in Table D.3. Every token name is the literal identifier that appears in the compiler’s token dump; the `KR_` prefix marks a keyword, `L_` and `D_` a left and right bracket, and `OP_` an operator (Section A.2).

Table D.3: Every token the lexer can emit, with its source lexeme and meaning. Numeric constants cover decimal, hexadecimal (`0x...`), and fixed-point literals with a decimal point and optional exponent.

Lexeme	Token	Meaning
<i>Keywords</i>		

(Table D.3, continued)

Lexeme	Token	Meaning
break	KR_BREAK	loop break
char	KR_CHAR	character type
const	KR_CONST	const qualifier
continue	KR_CONTINUE	loop continue
else	KR_ELSE	else branch
float	KR_FLOAT	fixed-point type
for	KR_FOR	for loop
if	KR_IF	if statement
int	KR_INT	integer type
return	KR_RETURN	return statement
struct	KR_STRUCT	structure type
void	KR_VOID	void type
while	KR_WHILE	while loop
<i>Operators</i>		
+	PLUS	addition
-	MINUS	subtraction / negation
*	ASTERISK	multiply / dereference
/	OP_DIJELI	division
%	OP_MOD	remainder
++	OP_INC	increment
--	OP_DEC	decrement
=	OP_PRIDRUZI	assignment
<	OP_LT	less than
<=	OP_LTE	less or equal
>	OP_GT	greater than
>=	OP_GTE	greater or equal
==	OP_EQ	equal
!=	OP_NEQ	not equal
!	OP_NEG	logical not
~	OP_TILDA	bitwise not
&&	OP_I	logical and
	OP_ILI	logical or
&	AMPERSAND	bitwise and / address-of
	OP_BIN_ILI	bitwise or
^	OP_BIN_XILI	bitwise xor
<i>Punctuation and literals</i>		
,	ZAREZ	comma
;	TOCKAZAREZ	semicolon
.	TOCKA	field selector
()	L_ZAGRADA D_ZAGRADA	parentheses
[]	L_UGL_ZAGRADA D_UGL_ZAGRADA	brackets
{ }	L_VIT_ZAGRADA D_VIT_ZAGRADA	braces
(name)	IDN	identifier
(number)	BROJ	numeric constant
(char)	ZNAK	character constant
(string)	NIZ_ZNAKOVA	string literal

These tokens are the parser’s entire input vocabulary; the next section shows how the grammar assembles them into the language the compiler actually accepts.

D.3 The accepted C subset

The grammar accepts a substantial fragment of C — considerably larger than the tinyC of the companion repository ([Appendix E](#)), though still a strict subset of the full language. [Table D.4](#) summarizes what is in and what is out. The language has four base types — void, char, int, and float — plus the const qualifier, pointers, fixed-size one-dimensional arrays, and structures. Functions take typed parameters and return a typed result. Control flow is if/else, while, and for, with break, continue, and return. Expressions span the full precedence ladder from assignment down through the logical, bitwise, equality, relational, additive, and multiplicative operators to casts, unary operators, and the postfix forms — array indexing, field selection, function calls, and post-increment.

What the subset leaves out is mostly what would complicate the back end without teaching anything new: there is no preprocessor, no typedef, no union or enum, no switch, do/while, or goto, no function pointers, and no multi-dimensional arrays. The float type is real but is represented in Q16.16 fixed point rather than IEEE 754 ([Section B.8](#)), which is why the language calls it float while the runtime treats it as a scaled integer.

Accepted	Not accepted
int, char, float, void	double, short, long, unsigned
const qualifier	volatile, static, extern
pointers, address-of, deref	function pointers
1-D arrays	multi-dimensional arrays
struct with fields	union, enum, bit-fields
functions with typed parameters	variadic functions
if/else, while, for	switch, do/while, goto
break, continue, return	labels
char and string literals	the preprocessor (#include, macros)
full operator precedence	comma operator in arbitrary position

Table D.4: What FRISCCc’s source language includes and excludes. The excluded constructs are omitted to keep the back end tractable, not because they raise new front-end questions.

D.4 Syntax

The parser specification declares the grammar in four parts: the list of non-terminals, the list of terminals (the token alphabet of [Table D.3](#)), the synchronization tokens used to resume after a syntax error — the semicolon and the closing brace, the two points at which

a statement or block reliably ends — and then the productions. Each production is a non-terminal followed by its alternatives, one per line.

The grammar is the standard C expression-precedence cascade. A translation unit is a sequence of external declarations, each either a function definition or a declaration. An expression is an assignment-expression, which is either a logical-or-expression or an lvalue assigned an assignment-expression; logical-or descends through logical-and, bitwise-or, bitwise-xor, bitwise-and, equality, relational, additive, and multiplicative expressions to cast-expressions, unary expressions, and finally postfix and primary expressions. This descent is exactly what encodes operator precedence and associativity into the parse tree without any precedence declarations. Statements are compound statements (a brace-enclosed list, optionally preceded by declarations), expression statements, the two branching forms, the three loop forms, and the jump statements. The full production set is the parser specification itself; [Chapter 4](#) walks the construction of the LR(1) tables from it, and [Section A.2](#) translates the non-terminal names used in this book.

D.5 Semantics

The semantics specification is the same grammar viewed as the scaffold for type checking. The semantic analyzer performs a post-order walk of the parse tree, and at each production it applies the typing judgement of [Chapter 5](#): a primary expression that is an identifier takes the type recorded for that name in the active scope; a binary operator requires operands of compatible types and yields a result type by the usual arithmetic conversions; an assignment requires its left operand to be an lvalue of a type the right operand converts to; a function call requires the argument types to match the declared parameter types; a return requires its expression to match the function's declared return type. Scopes are managed with a stack: entering a block pushes a fresh scope, leaving it pops, and a declaration inserts a name into the top scope. Loop and branch context is tracked so that a `break` or `continue` outside any loop is rejected. The analyzer's output is the parse tree decorated with a type at every expression node — the *typed* tree that the IR lowering of [Chapter 6](#) consumes. The judgements themselves, with their inference rules, are the body of [Chapter 5](#); the specification file fixes the productions they range over.

D.6 The source-language specification, in full

The three preceding sections described the source-language specifications; this section reproduces them. What follows is the literal content of the three files the front end is built from, with one editorial change: the non-terminal names, which in the shipped files are Croatian (the language of the original course material), are rendered here in the English the rest of the book uses, and the lexer's state, character-class, and action keywords are likewise given in English. The Croatian-to-English non-terminal mapping is in [Section A.2](#). The terminal token names — `KR_INT`, `OP_DIJELI`, `L_ZAGRADA`, and the rest — are left exactly

as the compiler emits them; their meanings are in Table D.3. Nothing else is altered: the structure, the rule order, and the regular expressions are the real specifications.

D.6.1 The lexer specification

The lexer file has four parts. It opens with *named character classes*, each a line {name} regex that abbreviates an alternation for reuse below; {anyChar}, for instance, names every printable character. The %X directive then declares the *exclusive states* (Table D.2), and %L lists the complete *lexical-unit* alphabet — the token names the rules may emit. The remainder is the rules. Each rule is a state-guarded regular expression, <state>regex, followed by a brace block of actions: the first line names the token to emit, or a lone - to emit nothing; ENTER_STATE s switches the scanner to state s; NEWLINE records a line break for diagnostics; and GO_BACK n rolls the read position back by n characters so a delimiter is re-scanned in the state just entered.

```

1  {letter} a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|_
   ↳ S|T|U|V|W|X|Y|Z
2  {digit} 0|1|2|3|4|5|6|7|8|9
3  {hexDigit} {digit}|a|b|c|d|e|f|A|B|C|D|E|F
4  {whitespace} \t|\n|\\
5  {exponent} (e|E)($|+|-){digit}{digit}*
6  {anyChar} \(\)|\{|\}|\||\*|\||\$|\t|\n|\\|!|"|#|%&|'|+|,|-|.|/|0|1|2|3|4|5|6|7|8|9|:|;|<|=|>|?|_
   ↳ @|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|[\]^_`|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|
   ↳ |p|q|r|s|t|u|v|w|x|y|z|~
7  {anyButQuoteOrNewline} \(\)|\{|\}|\||\*|\||\$|\t|\n|\\|!|"|#|%&|'|+|,|-|.|/|0|1|2|3|4|5|6|7|8|9|:|;|_
   ↳ |<|=|>|?|@|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|[\]^_`|a|b|c|d|e|f|g|h|i|j|k|
   ↳ |l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|~
8  {anyButQuoteNewLineTab} \(\)|\{|\}|\||\*|\||\$|\t|\n|\\|!|"|#|%&|'|+|,|-|.|/|0|1|2|3|4|5|6|7|8|9|:|;|_
   ↳ |=|>|?|@|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|[\]^_`|a|b|c|d|e|f|g|h|i|j|k|
   ↳ |l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|~
9  {anyButNewLineTab} \(\)|\{|\}|\||\*|\||\$|\t|\n|\\|!|"|#|%&|'|+|,|-|.|/|0|1|2|3|4|5|6|7|8|9|:|;|<|=|>|_
   ↳ |?|@|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|[\]^_`|a|b|c|d|e|f|g|h|i|j|k|l|m|
   ↳ |n|o|p|q|r|s|t|u|v|w|x|y|z|~
10 %X S_start S_blockComment S_lineComment S_string
11 %L IDN BROJ ZNAK NIZ_ZNAKOVA KR_BREAK KR_CHAR KR_CONST KR_CONTINUE KR_ELSE KR_FLOAT KR_FOR KR_IF
   ↳ KR_INT KR_RETURN KR_STRUCT KR_VOID KR_WHILE PLUS OP_INC MINUS OP_DEC ASTERISK OP_DIJELI
   ↳ OP_MOD OP_PRIDRUZI OP_LT OP_LTE OP_GT OP_GTE OP_EQ OP_NEQ OP_NEG OP_TILDA OP_I OP_ILI
   ↳ AMPERSAND OP_BIN_ILI OP_BIN_XILI ZAREZ TOCKAZAREZ TOCKA L_ZAGRADA D_ZAGRADA L_UGL_ZAGRADA
   ↳ D_UGL_ZAGRADA L_VIT_ZAGRADA D_VIT_ZAGRADA
12 <S_start>\t|\\
13 {
14 -
15 }
16 <S_start>\n
17 {
18 -
19 NEWLINE
20 }
21 <S_start>//

```

```
22 {
23 -
24 ENTER_STATE S_lineComment
25 }
26 <S_lineComment>\n
27 {
28 -
29 NEWLINE
30 ENTER_STATE S_start
31 }
32 <S_lineComment>{anyChar}
33 {
34 -
35 }
36 <S_start>/\*
37 {
38 -
39 ENTER_STATE S_blockComment
40 }
41 <S_blockComment>\*/
42 {
43 -
44 ENTER_STATE S_start
45 }
46 <S_blockComment>\n
47 {
48 -
49 NEWLINE
50 }
51 <S_blockComment>{anyChar}
52 {
53 -
54 }
55 <S_start>"
56 {
57 -
58 ENTER_STATE S_string
59 GO_BACK 0
60 }
61 <S_string>"({anyButDquoteOrNewline}|\\"")*"
62 {
63 NIZ_ZNAKOVA
64 ENTER_STATE S_start
65 }
66 <S_start>break
67 {
68 KR_BREAK
69 }
70 <S_start>char
71 {
72 KR_CHAR
73 }
74 <S_start>const
```

```

75 {
76   KR_CONST
77 }
78 <S_start>continue
79 {
80   KR_CONTINUE
81 }
82 <S_start>else
83 {
84   KR_ELSE
85 }
86 <S_start>float
87 {
88   KR_FLOAT
89 }
90 <S_start>for
91 {
92   KR_FOR
93 }
94 <S_start>if
95 {
96   KR_IF
97 }
98 <S_start>int
99 {
100   KR_INT
101 }
102 <S_start>return
103 {
104   KR_RETURN
105 }
106 <S_start>struct
107 {
108   KR_STRUCT
109 }
110 <S_start>void
111 {
112   KR_VOID
113 }
114 <S_start>while
115 {
116   KR_WHILE
117 }
118 <S_start>(_|{letter})(_|{letter}|{digit})*
119 {
120   IDN
121 }
122 <S_start>{digit}{digit}*
123 {
124   BROJ
125 }
126 <S_start>0(X|x){hexDigit}{hexDigit}*
127 {

```

```

128     BROJ
129   }
130   <S_start>{digit}{digit}*.{digit}*($|{exponent})
131   {
132     BROJ
133   }
134   <S_start>{digit}*.{digit}{digit}*($|{exponent})
135   {
136     BROJ
137   }
138   <S_start>'{anyButQuoteNewlineTab}'
139   {
140     ZNAK
141   }
142   <S_start>'\\{anyButNewlineTab}'
143   {
144     ZNAK
145   }
146   <S_start>++
147   {
148     OP_INC
149   }
150   <S_start>--
151   {
152     OP_DEC
153   }
154   <S_start>+
155   {
156     PLUS
157   }
158   <S_start>-
159   {
160     MINUS
161   }
162   <S_start>\*
163   {
164     ASTERISK
165   }
166   <S_start>/
167   {
168     OP_DIJELI
169   }
170   <S_start>%
171   {
172     OP_MOD
173   }
174   <S_start>=
175   {
176     OP_PRIDRUZI
177   }
178   <S_start><
179   {
180     OP_LT

```

```

181     }
182     <S_start><=
183     {
184         OP_LTE
185     }
186     <S_start>>
187     {
188         OP_GT
189     }
190     <S_start>>=
191     {
192         OP_GTE
193     }
194     <S_start>==
195     {
196         OP_EQ
197     }
198     <S_start>!=
199     {
200         OP_NEQ
201     }
202     <S_start>!
203     {
204         OP_NEG
205     }
206     <S_start>~
207     {
208         OP_TILDA
209     }
210     <S_start>&&
211     {
212         OP_I
213     }
214     <S_start>\\|
215     {
216         OP_ILI
217     }
218     <S_start>&
219     {
220         AMPERSAND
221     }
222     <S_start>\\|
223     {
224         OP_BIN_ILI
225     }
226     <S_start>^
227     {
228         OP_BIN_XILI
229     }
230     <S_start>,
231     {
232         ZAREZ
233     }

```



```

234 <S_start>;
235 {
236   TOCKAZAREZ
237 }
238 <S_start>.
239 {
240   TOCKA
241 }
242 <S_start>\(
243 {
244   L_ZAGRADA
245 }
246 <S_start>\)
247 {
248   D_ZAGRADA
249 }
250 <S_start>\{
251 {
252   L_VIT_ZAGRADA
253 }
254 <S_start>\}
255 {
256   D_VIT_ZAGRADA
257 }
258 <S_start>[
259 {
260   L_UGL_ZAGRADA
261 }
262 <S_start>]
263 {
264   D_UGL_ZAGRADA
265 }

```

D.6.2 The parser specification

The parser file opens with three directives: %V lists the non-terminals (the grammar variables), %T lists the terminals (the token alphabet of [Table D.3](#)), and %Syn names the *synchronization* tokens — the semicolon and the closing brace — at which the parser resumes after a syntax error. The productions follow: each non-terminal appears on its own line, and each of its alternatives is an indented line beneath it, a sequence of non-terminals and terminals. From these the canonical LR(1) tables of [Chapter 4](#) are built.

```

1  %V <translation_unit> <external_declaration> <declaration> <function_definition>
   ↳ <declaration_specifiers> <declarator> <primary_expression> <expression> <postfix_expression>
   ↳ <argument_list> <assignment_expression> <unary_expression> <unary_operator> <cast_expression>
   ↳ <type_name> <multiplicative_expression> <additive_expression> <relational_expression>
   ↳ <equality_expression> <bitwise_and_expression> <bitwise_xor_expression> <bitwise_or_expression>
   ↳ <logical_and_expression> <logical_or_expression> <type_specifier> <init_declarator_list>
   ↳ <init_declarator> <initializer> <struct_specifier> <struct_declaration_list>
   ↳ <struct_declaration> <specifier_qualifier_list> <struct_declarator_list> <struct_declarator>
   ↳ <pointer> <direct_declarator> <parameter_list> <parameter_declaration>
   ↳ <assignment_expression_list> <statement> <compound_statement> <expression_statement>
   ↳ <branch_statement> <loop_statement> <jump_statement> <statement_list> <declaration_list>
2  %T IDN BROJ ZNAK NIZ_ZNAKOVA KR_BREAK KR_CHAR KR_CONST KR_CONTINUE KR_ELSE KR_FLOAT KR_FOR KR_IF
   ↳ KR_INT KR_RETURN KR_STRUCT KR_VOID KR_WHILE PLUS OP_INC MINUS OP_DEC ASTERISK OP_DIJELI OP_MOD
   ↳ OP_PRIDRUZI OP_LT OP_LTE OP_GT OP_GTE OP_EQ OP_NEQ OP_NEG OP_TILDA OP_I OP_ILI AMPERSAND
   ↳ OP_BIN_ILI OP_BIN_XILI ZAREZ TOCKAZAREZ TOCKA L_ZAGRADA D_ZAGRADA L_UGL_ZAGRADA D_UGL_ZAGRADA
   ↳ L_VIT_ZAGRADA D_VIT_ZAGRADA
3  %Syn TOCKAZAREZ D_VIT_ZAGRADA
4  <primary_expression>
5      IDN
6      BROJ
7      ZNAK
8      NIZ_ZNAKOVA
9      L_ZAGRADA <expression> D_ZAGRADA
10 <postfix_expression>
11 <primary_expression>
12 <postfix_expression> L_UGL_ZAGRADA <expression> D_UGL_ZAGRADA
13 <postfix_expression> L_ZAGRADA D_ZAGRADA
14 <postfix_expression> L_ZAGRADA <argument_list> D_ZAGRADA
15 <postfix_expression> TOCKA IDN
16 <postfix_expression> OP_INC
17 <postfix_expression> OP_DEC
18 <argument_list>
19 <assignment_expression>
20 <argument_list> ZAREZ <assignment_expression>
21 <unary_expression>
22 <postfix_expression>
23 OP_INC <unary_expression>
24 OP_DEC <unary_expression>
25 <unary_operator> <cast_expression>
26 <unary_operator>
27 AMPERSAND
28 ASTERISK
29 PLUS
30 MINUS
31 OP_TILDA
32 OP_NEG
33 <cast_expression>
34 <unary_expression>
35 L_ZAGRADA <type_name> D_ZAGRADA <cast_expression>
36 <multiplicative_expression>
37 <cast_expression>
38 <multiplicative_expression> ASTERISK <cast_expression>
39 <multiplicative_expression> OP_DIJELI <cast_expression>

```

```

40     <multiplicative_expression> OP_MOD <cast_expression>
41 <additive_expression>
42     <multiplicative_expression>
43     <additive_expression> PLUS <multiplicative_expression>
44     <additive_expression> MINUS <multiplicative_expression>
45 <relational_expression>
46     <additive_expression>
47     <relational_expression> OP_LT <additive_expression>
48     <relational_expression> OP_GT <additive_expression>
49     <relational_expression> OP_LTE <additive_expression>
50     <relational_expression> OP_GTE <additive_expression>
51 <equality_expression>
52     <relational_expression>
53     <equality_expression> OP_EQ <relational_expression>
54     <equality_expression> OP_NEQ <relational_expression>
55 <bitwise_and_expression>
56     <equality_expression>
57     <bitwise_and_expression> AMPERSAND <equality_expression>
58 <bitwise_xor_expression>
59     <bitwise_and_expression>
60     <bitwise_xor_expression> OP_BIN_XILI <bitwise_and_expression>
61 <bitwise_or_expression>
62     <bitwise_xor_expression>
63     <bitwise_or_expression> OP_BIN_ILI <bitwise_xor_expression>
64 <logical_and_expression>
65     <bitwise_or_expression>
66     <logical_and_expression> OP_I <bitwise_or_expression>
67 <logical_or_expression>
68     <logical_and_expression>
69     <logical_or_expression> OP_ILI <logical_and_expression>
70 <assignment_expression>
71     <logical_or_expression>
72     <unary_expression> OP_PRIDRUZI <assignment_expression>
73 <expression>
74     <assignment_expression>
75     <expression> ZAREZ <assignment_expression>
76 <declaration>
77     <declaration_specifiers> TOCKAZAREZ
78     <declaration_specifiers> <init_declarator_list> TOCKAZAREZ
79 <declaration_specifiers>
80     <type_specifier>
81     <type_specifier> <declaration_specifiers>
82     KR_CONST
83     KR_CONST <declaration_specifiers>
84 <init_declarator_list>
85     <init_declarator>
86     <init_declarator_list> ZAREZ <init_declarator>
87 <init_declarator>
88     <declarator>
89     <declarator> OP_PRIDRUZI <initializer>
90 <type_specifier>
91     KR_VOID
92     KR_CHAR

```

```

93  KR_INT
94  KR_FLOAT
95  <struct_specifier>
96  <struct_specifier>
97  KR_STRUCT IDN L_VIT_ZAGRADA <struct_declaration_list> D_VIT_ZAGRADA
98  KR_STRUCT L_VIT_ZAGRADA <struct_declaration_list> D_VIT_ZAGRADA
99  KR_STRUCT IDN
100 <struct_declaration_list>
101 <struct_declaration>
102 <struct_declaration_list> <struct_declaration>
103 <struct_declaration>
104 <specifier_qualifier_list> <struct_declarator_list> TOCKAZAREZ
105 <specifier_qualifier_list>
106 <type_specifier> <specifier_qualifier_list>
107 <type_specifier>
108 KR_CONST <specifier_qualifier_list>
109 KR_CONST
110 <struct_declarator_list>
111 <struct_declarator>
112 <struct_declarator_list> ZAREZ <struct_declarator>
113 <struct_declarator>
114 <declarator>
115 <declarator>
116 <pointer> <direct_declarator>
117 <direct_declarator>
118 <direct_declarator>
119 IDN
120 <direct_declarator> L_UGL_ZAGRADA <logical_or_expression> D_UGL_ZAGRADA
121 <direct_declarator> L_UGL_ZAGRADA D_UGL_ZAGRADA
122 <direct_declarator> L_ZAGRADA <parameter_list> D_ZAGRADA
123 <pointer>
124 ASTERISK
125 ASTERISK KR_CONST
126 <parameter_list>
127 <parameter_declaration>
128 <parameter_list> ZAREZ <parameter_declaration>
129 <parameter_declaration>
130 <declaration_specifiers> <declarator>
131 <declaration_specifiers>
132 <type_name>
133 <specifier_qualifier_list>
134 <specifier_qualifier_list> <pointer>
135 <initializer>
136 <assignment_expression>
137 L_VIT_ZAGRADA <assignment_expression_list> D_VIT_ZAGRADA
138 L_VIT_ZAGRADA <assignment_expression_list> ZAREZ D_VIT_ZAGRADA
139 <assignment_expression_list>
140 <assignment_expression>
141 <assignment_expression_list> ZAREZ <assignment_expression>
142 <statement>
143 <compound_statement>
144 <expression_statement>
145 <branch_statement>

```

```

146     <loop_statement>
147     <jump_statement>
148     <compound_statement>
149     L_VIT_ZAGRADA D_VIT_ZAGRADA
150     L_VIT_ZAGRADA <statement_list> D_VIT_ZAGRADA
151     L_VIT_ZAGRADA <declaration_list> D_VIT_ZAGRADA
152     L_VIT_ZAGRADA <declaration_list> <statement_list> D_VIT_ZAGRADA
153     <declaration_list>
154     <declaration>
155     <declaration_list> <declaration>
156     <statement_list>
157     <statement>
158     <statement_list> <statement>
159     <expression_statement>
160     TOCKAZAREZ
161     <expression> TOCKAZAREZ
162     <branch_statement>
163     KR_IF L_ZAGRADA <expression> D_ZAGRADA <statement>
164     KR_IF L_ZAGRADA <expression> D_ZAGRADA <statement> KR_ELSE <statement>
165     <loop_statement>
166     KR_WHILE L_ZAGRADA <expression> D_ZAGRADA <statement>
167     KR_FOR L_ZAGRADA <expression_statement> <expression_statement> D_ZAGRADA <statement>
168     KR_FOR L_ZAGRADA <expression_statement> <expression_statement> <expression> D_ZAGRADA <statement>
169     <jump_statement>
170     KR_CONTINUE TOCKAZAREZ
171     KR_BREAK TOCKAZAREZ
172     KR_RETURN TOCKAZAREZ
173     KR_RETURN <expression> TOCKAZAREZ
174     <translation_unit>
175     <external_declaration>
176     <translation_unit> <external_declaration>
177     <external_declaration>
178     <function_definition>
179     <declaration>
180     <function_definition>
181     <declaration_specifiers> <declarator> <declaration_list> <compound_statement>
182     <declaration_specifiers> <declarator> <compound_statement>
183     <declarator> <declaration_list> <compound_statement>
184     <declarator> <compound_statement>

```

D.6.3 The semantics specification

The semantics file is the same grammar written as plain BNF — a non-terminal, `::=`, and its alternatives separated by `|` — and it is the production set the semantic analyzer walks while attaching the typing judgements of [Chapter 5](#). It overlaps almost entirely with the parser grammar above; the few differences are alternate spellings of the same terminals (the file writes `OP_PUTA` for multiplication and `OP_BIN_I` for bitwise-and where the parser writes `ASTERISK` and `AMPERSAND`), which the front end reconciles.

```

1  <primary_expression> ::= IDN
2      | BROJ
3      | ZNAK
4      | NIZ_ZNAKOVA
5      | L_ZAGRADA <expression> D_ZAGRADA
6  <postfix_expression> ::= <primary_expression>
7      | <postfix_expression> L_UGL_ZAGRADA <expression> D_UGL_ZAGRADA
8      | <postfix_expression> L_ZAGRADA D_ZAGRADA
9      | <postfix_expression> L_ZAGRADA <argument_list> D_ZAGRADA
10     | <postfix_expression> TOCKA IDN
11     | <postfix_expression> OP_INC
12     | <postfix_expression> OP_DEC
13 <argument_list> ::= <assignment_expression>
14     | <argument_list> ZAREZ <assignment_expression>
15 <unary_expression> ::= <postfix_expression>
16     | OP_INC <unary_expression>
17     | OP_DEC <unary_expression>
18     | <unary_operator> <cast_expression>
19     | AMPERSAND <cast_expression>
20     | ASTERISK <cast_expression>
21 <unary_operator> ::= PLUS
22     | MINUS
23     | OP_TILDA
24     | OP_NEG
25 <cast_expression> ::= <unary_expression>
26     | L_ZAGRADA <type_name> D_ZAGRADA <cast_expression>
27 <type_name> ::= <specifier_qualifier_list>
28     | <specifier_qualifier_list> <pointer>
29 <specifier_qualifier_list> ::= <type_specifier>
30     | <specifier_qualifier_list> <type_specifier>
31     | <specifier_qualifier_list> KR_CONST
32 <type_specifier> ::= KR_VOID
33     | KR_CHAR
34     | KR_INT
35     | KR_FLOAT
36     | <struct_specifier>
37 <pointer> ::= ASTERISK
38     | ASTERISK KR_CONST
39     | ASTERISK <pointer>
40     | ASTERISK KR_CONST <pointer>
41 <struct_specifier> ::= KR_STRUCT IDN L_VIT_ZAGRADA <struct_declaration_list> D_VIT_ZAGRADA
42     | KR_STRUCT L_VIT_ZAGRADA <struct_declaration_list> D_VIT_ZAGRADA
43     | KR_STRUCT IDN
44 <struct_declaration_list> ::= <struct_declaration>
45     | <struct_declaration_list> <struct_declaration>
46 <struct_declaration> ::= <specifier_qualifier_list> <struct_declarator_list> TOCKAZAREZ
47 <struct_declarator_list> ::= <struct_declarator>
48     | <struct_declarator_list> ZAREZ <struct_declarator>
49 <struct_declarator> ::= IDN
50     | IDN L_UGL_ZAGRADA BROJ D_UGL_ZAGRADA
51     | <pointer> IDN
52     | <pointer> IDN L_UGL_ZAGRADA BROJ D_UGL_ZAGRADA
53 <multiplicative_expression> ::= <cast_expression>

```

```

54 | <multiplicative_expression> OP_PUTA <cast_expression>
55 | <multiplicative_expression> OP_DIJELI <cast_expression>
56 | <multiplicative_expression> OP_MOD <cast_expression>
57 <additive_expression> ::= <multiplicative_expression>
58 | <additive_expression> PLUS <multiplicative_expression>
59 | <additive_expression> MINUS <multiplicative_expression>
60 <relational_expression> ::= <additive_expression>
61 | <relational_expression> OP_LT <additive_expression>
62 | <relational_expression> OP_GT <additive_expression>
63 | <relational_expression> OP_LTE <additive_expression>
64 | <relational_expression> OP_GTE <additive_expression>
65 <equality_expression> ::= <relational_expression>
66 | <equality_expression> OP_EQ <relational_expression>
67 | <equality_expression> OP_NEQ <relational_expression>
68 <bitwise_and_expression> ::= <equality_expression>
69 | <bitwise_and_expression> OP_BIN_I <equality_expression>
70 <bitwise_xor_expression> ::= <bitwise_and_expression>
71 | <bitwise_xor_expression> OP_BIN_XILI <bitwise_and_expression>
72 <bitwise_or_expression> ::= <bitwise_xor_expression>
73 | <bitwise_or_expression> OP_BIN_ILI <bitwise_xor_expression>
74 <logical_and_expression> ::= <bitwise_or_expression>
75 | <logical_and_expression> OP_I <bitwise_or_expression>
76 <logical_or_expression> ::= <logical_and_expression>
77 | <logical_or_expression> OP_ILI <logical_and_expression>
78 <assignment_expression> ::= <logical_or_expression>
79 | <postfix_expression> OP_PRIDRUZI <assignment_expression>
80 <expression> ::= <assignment_expression>
81 | <expression> ZAREZ <assignment_expression>
82 <compound_statement> ::= L_VIT_ZAGRADA <statement_list> D_VIT_ZAGRADA
83 | L_VIT_ZAGRADA <declaration_list> <statement_list> D_VIT_ZAGRADA
84 <statement_list> ::= <statement>
85 | <statement_list> <statement>
86 <statement> ::= <compound_statement>
87 | <expression_statement>
88 | <branch_statement>
89 | <loop_statement>
90 | <jump_statement>
91 <expression_statement> ::= TOCKAZAREZ
92 | <expression> TOCKAZAREZ
93 <branch_statement> ::= KR_IF L_ZAGRADA <expression> D_ZAGRADA <statement>
94 | KR_IF L_ZAGRADA <expression> D_ZAGRADA <statement> KR_ELSE <statement>
95 <loop_statement> ::= KR_WHILE L_ZAGRADA <expression> D_ZAGRADA <statement>
96 | KR_FOR L_ZAGRADA <expression_statement> <expression_statement> D_ZAGRADA <statement>
97 | KR_FOR L_ZAGRADA <expression_statement> <expression_statement> <expression> D_ZAGRADA
98   ↪ <statement>
99 <jump_statement> ::= KR_CONTINUE TOCKAZAREZ
100 | KR_BREAK TOCKAZAREZ
101 | KR_RETURN TOCKAZAREZ
102 | KR_RETURN <expression> TOCKAZAREZ
103 <translation_unit> ::= <external_declaration>
104 | <translation_unit> <external_declaration>
105 <external_declaration> ::= <function_definition>
106 | <declaration>

```

```

106 <function_definition> ::= <type_name> IDN L_ZAGRADA KR_VOID D_ZAGRADA <compound_statement>
107 | <type_name> IDN L_ZAGRADA <parameter_list> D_ZAGRADA <compound_statement>
108 <parameter_list> ::= <parameter_declaration>
109 | <parameter_list> ZAREZ <parameter_declaration>
110 <parameter_declaration> ::= <type_name> IDN
111 | <type_name> IDN L_UGL_ZAGRADA D_UGL_ZAGRADA
112 <declaration_list> ::= <declaration>
113 | <declaration_list> <declaration>
114 <declaration> ::= <type_name> <init_declarator_list> TOCKAZAREZ
115 <init_declarator_list> ::= <init_declarator>
116 | <init_declarator_list> ZAREZ <init_declarator>
117 <init_declarator> ::= <direct_declarator>
118 | <direct_declarator> OP_PRIDRUZI <initializer>
119 <direct_declarator> ::= IDN
120 | IDN L_UGL_ZAGRADA BROJ D_UGL_ZAGRADA
121 | IDN L_ZAGRADA KR_VOID D_ZAGRADA
122 | IDN L_ZAGRADA <parameter_list> D_ZAGRADA
123 <initializer> ::= <assignment_expression>
124 | L_VIT_ZAGRADA <assignment_expression_list> D_VIT_ZAGRADA
125 <assignment_expression_list> ::= <assignment_expression>
126 | <assignment_expression_list> ZAREZ <assignment_expression>

```

D.7 The intermediate representation

The remainder of the appendix is the grammar of the textual IR, the form the compiler prints after lowering and reads back into the interpreter and the bytecode VM. It is a typed three-address representation: every value carries an explicit type, every computation names its result in a temporary or a memory slot, and control flow is an explicit graph of basic blocks.

D.7.1 Lexical structure

Newlines are significant: they separate instructions and terminate the header lines of modules, functions, and blocks. A temporary is the letter `t` followed by an integer, as in `t0`, `t17`. An integer literal is an optional minus sign and decimal digits. A float literal is an optional minus sign, digits, a point, and digits. A character literal is a quoted character or escape. An identifier is a letter followed by letters, digits, and underscores, and names functions, globals, slots, struct types, and block labels. A constant is written with a leading `#` and an explicit type, as in `#5:int32` or `#5.5:float`, or as the typed `null` for a pointer.

D.7.2 Module structure

A module is delimited by `.program` and `.endprogram` and contains, in order, an optional globals section, zero or more named struct-type definitions, and the function definitions. The globals section, introduced by `.globals`, lists global variables, each a name, a type, and an optional initializing constant. A struct-type definition, introduced by `.type struct`, lists the fields of a named structure, each with a type and an explicit byte offset — the layout is fixed in the IR, not left to the back end.

D.7.3 Functions and basic blocks

A function definition, introduced by `.func`, declares a name, a parameter list, and a return type, then carries three sections. The `.frame` line records the size in bytes of the function's locals and the frame's alignment. The `.slots` section is the slot table: one entry per parameter, local, or spill, each naming the slot, its kind, its byte offset within the frame, and its type — this is the data the code generator turns into frame-pointer-relative addresses (Chapter 9). The `.blocks` section is the control-flow graph: a sequence of basic blocks, each a label followed by zero or more instructions and exactly one terminator. The function ends with `.endfunc`.

D.7.4 Instruction grammar

A non-terminating instruction is one of three forms. An *assignment* names a temporary and gives it a right-hand side: $t_n = \text{rhs}$. A *store* writes a typed value to a memory address: `store addr, value : type`. A *void call* invokes a function for its effect, discarding any result.

A right-hand side is any value-producing form: the address of a named symbol (`addr_of_symbol local:x, param:, or global:`), the address of an array element (`addr_index base, index, elemSize`), the address of a struct field (`addr_field base, Type.field`), a typed load (`load addr : type`), a binary operation (`add, sub, mul, div, mod, and, or, xor, shl, shr`), a comparison (`cmp_eq, cmp_ne, cmp_lt, cmp_le, cmp_gt, cmp_ge`, each yielding `bool`), a unary operation (`neg, not`), an increment or decrement form (`preinc, postinc, predec, postdec`), a cast (`trunc, sext, zext, ptrcast, itof, ftoi`), a value-returning call (`call func:f(...) : type`), or a constant. Every binary, unary, comparison, and cast form carries the explicit result type after a colon, which is what makes the IR *typed*: a pass never has to re-derive the type of a value, because the value states it.

Each block ends in exactly one terminator: a conditional branch (`br cond, trueLabel, falseLabel`), an unconditional jump (`jmp label`), or a return (`ret` or `ret value`).

D.7.5 Type grammar

The IR types are `void`; the scalar types `int32`, `char`, `uchar`, `float`, and `bool`; the pointer type `ptr<T>` for any type `T`; the array type `array<T,N>` for element type `T` and length `N`; and the named structure type `struct Name`. The source language's `int` lowers to `int32` and its `float` to the Q16.16 `float`; `bool` is the result type of comparisons and the condition type of branches, and has no direct source spelling.

D.7.6 Well-formedness conditions

Not every string the grammar admits is a legal program. A function body is *well-formed* only when it satisfies the eight invariants below. The IR verifier that runs at the entry to optimization and, optionally, after every pass (Chapter 6) mechanically checks the structural and typing conditions — invariants 1, 2, 3, 4, and 6; the remaining ones, invariants 5, 7, and 8, are properties the lowering establishes by construction and the proofs assume. Taken together they are the conditions the semantic-preservation proofs of Appendix G assume of their inputs and require their outputs to maintain (the framework's well-formedness obligation).

1. Every basic block has exactly one terminator, and it is the last instruction in the block.
2. Within each basic block, every temporary is defined before it is used.
3. Every opcode's operand and result types are consistent with the type catalogue of Section D.7.5.
4. Every block label is unique within the function.
5. The entry block has no incoming edges.
6. Every branch and jump target is a label in the same function.
7. Every `ret` carries the type declared in the function signature, and every `void` function reaches a bare `ret` on every path.
8. The slot table covers every `addr_of_symbol` target, and every listed slot is referenced.

A program is well-formed when every function in it is. Invariants 1, 4, 5, and 6 fix the shape of the control-flow graph; invariants 2 and 3 fix the data flow and typing; invariants 7 and 8 fix the boundary between a function and its frame. Together they are what let an optimization pass reason locally and still be sure it has not produced a program that means something different — or nothing at all.

D.8 The IR specification, in full

The subsections above walked the IR grammar a piece at a time; this is the whole of it, the single file the IR printer and parser are built from. Unlike the source-language specifications, it needs no translation — the IR’s keywords (`.program`, `.func`, `.blocks`, `addr_of_symbol`, and the rest) are English already. It is written as a BNF grammar, with `::=` for productions, `|` for alternatives, braces `{ ... }` for zero-or-more repetition, brackets `[ ... ]` for optional parts, and lines beginning with a semicolon as comments that are part of the listing rather than the grammar.

```

1  ; =====
2  ; IR Grammar (BNF) – single-file, copy/paste ready
3  ; Notes:
4  ; - This grammar is intentionally strict and fully typed.
5  ; - Newlines (NL) are significant at statement boundaries.
6  ; - Comments here are part of this listing (for readability).
7  ; =====
8
9  Program
10 ::= ".program" NL
11     { TopLevel }
12     ".endprogram" NL? ;
13
14 ; -----
15 ; Top-level declarations
16 ; -----
17
18 TopLevel
19 ::= GlobalDecl
20     | TypeDef
21     | FuncDef ;
22
23 ; Globals section contains zero or more global variables.
24 GlobalDecl
25 ::= ".globals" NL
26     { GlobalVar } ;
27
28 ; A global variable may optionally have an initializer constant.
29 GlobalVar
30 ::= "global" Ident ":" Type [ "=" Const ] NL ;
31
32 ; Named struct type definitions.
33 TypeDef
34 ::= ".type" "struct" Ident "{" NL
35     { StructField }
36     "}" NL ;
37
38 ; Each field has a name, a type, and an explicit byte offset.
39 StructField
40 ::= Ident ":" Type "@" Int NL ;
41

```

```

42 ; -----
43 ; Functions
44 ; -----
45
46 ; Function definition with explicit return type.
47 ; Return type may be "void" (then "ret" has no value).
48 FuncDef
49   ::= ".func" Ident "(" [ ParamList ] ")" ":" Type NL
50       FrameDecl
51       SlotsDecl
52       BlocksDecl
53       ".endfunc" NL ;
54
55 ; Comma-separated parameter list.
56 ParamList
57   ::= Param { "," Param } ;
58
59 Param
60   ::= Ident ":" Type ;
61
62 ; Stack frame metadata.
63 FrameDecl
64   ::= ".frame" "locals" "=" Int "bytes" "align" "=" Int NL ;
65
66 ; Slot table (params/locals/spills) with explicit byte offsets.
67 SlotsDecl
68   ::= ".slots" NL
69       { SlotEntry } ;
70
71 SlotEntry
72   ::= SlotKind Ident "@" Int ":" Type NL ;
73
74 SlotKind
75   ::= "param" | "local" | "spill" ;
76
77 ; -----
78 ; Control-flow graph (basic blocks)
79 ; -----
80
81 BlocksDecl
82   ::= ".blocks" NL
83       { Block } ;
84
85 ; A basic block: label + instructions + exactly one terminator.
86 Block
87   ::= Label ":" NL
88       { Instr NL }
89       Terminator NL ;
90
91 Label
92   ::= Ident ;           ; e.g., L0, L1, loop_body, etc.
93
94 ; -----

```

```

95 ; Instructions
96 ; -----
97
98 ; Non-terminating instructions.
99 Instr
100 ::= AssignInstr
101 | StoreInstr
102 | VoidCallInstr ;
103
104 ; Temp assignment: tN = <rhs>
105 AssignInstr
106 ::= Temp "=" Rhs ;
107
108 ; Typed store to memory.
109 ; Interpretation: store <addr>, <value> : <valueType>
110 StoreInstr
111 ::= "store" Value "," Value ":" Type ;
112
113 ; Call returning no value.
114 VoidCallInstr
115 ::= "call" "func:" Ident "(" [ ArgList ] ")" ":" "void" ;
116
117 ; -----
118 ; Terminators
119 ; -----
120
121 Terminator
122 ::= BrTerm
123 | JmpTerm
124 | RetTerm ;
125
126 ; Conditional branch:
127 ; br <cond>, <trueLabel>, <falseLabel>
128 BrTerm
129 ::= "br" Value "," Label "," Label ;
130
131 ; Unconditional jump:
132 ; jmp <label>
133 JmpTerm
134 ::= "jmp" Label ;
135
136 ; Return:
137 ; ret
138 ; ret <value>
139 RetTerm
140 ::= "ret" [ Value ] ;
141
142 ; -----
143 ; Calls and arguments
144 ; -----
145
146 ArgList
147 ::= Value { "," Value } ;

```

```

148
149 ; -----
150 ; RHS (right-hand side) expressions
151 ; -----
152
153 ; Any expression that yields a value (temp or constant).
154 Rhs
155 ::= AddrOfSymbol
156    | AddrIndex
157    | AddrField
158    | Load
159    | BinOp
160    | CmpOp
161    | Call
162    | UnaryOp
163    | IncDecOp
164    | CastOp
165    | Const ;
166
167 ; Address of a named symbol (local/param/global).
168 AddrOfSymbol
169 ::= "addr_of_symbol" SymbolRef ;
170
171 SymbolRef
172 ::= ("local:" | "param:" | "global:") Ident ;
173
174 ; Address of array element:
175 ;   addr_index <baseAddr>, <indexValue>, <elemSizeBytes>
176 AddrIndex
177 ::= "addr_index" Value "," Value "," Int ;
178
179 ; Address of a struct field:
180 ;   addr_field <baseAddr>, <structType>.<fieldName>
181 AddrField
182 ::= "addr_field" Value "," Ident "." Ident ;
183
184 ; Typed load from memory:
185 ;   load <addr> : <type>
186 Load
187 ::= "load" Value ":" Type ;
188
189 ; -----
190 ; Arithmetic / bitwise operations
191 ; -----
192
193 ; Binary operations are typed and return the given type.
194 BinOp
195 ::= BinOpName Value "," Value ":" Type ;
196
197 BinOpName
198 ::= "add" | "sub" | "mul" | "div" | "mod"
199    | "and" | "or" | "xor"
200    | "shl" | "shr" ;

```

```

201
202 ; -----
203 ; Comparisons
204 ; -----
205
206 ; Comparisons return bool.
207 CmpOp
208   ::= CmpOpName Value "," Value ":" "bool" ;
209
210 CmpOpName
211   ::= "cmp_eq" | "cmp_ne"
212      | "cmp_lt" | "cmp_le"
213      | "cmp_gt" | "cmp_ge" ;
214
215 ; -----
216 ; Unary operations
217 ; -----
218
219 UnaryOp
220   ::= UnaryOpName Value ":" Type ;
221
222 UnaryOpName
223   ::= "neg" | "not" ;
224
225 ; -----
226 ; Increment / decrement forms
227 ; -----
228
229 ; These are value-producing forms (typically used on lvalues lowered to loads/stores).
230 IncDecOp
231   ::= IncDecName Value ":" Type ;
232
233 IncDecName
234   ::= "preinc" | "postinc" | "predec" | "postdec" ;
235
236 ; -----
237 ; Function call producing a value
238 ; -----
239
240 Call
241   ::= "call" "func:" Ident "(" [ ArgList ] ")" ":" Type ;
242
243 ; -----
244 ; Casts / conversions
245 ; -----
246
247 CastOp
248   ::= CastName Value ":" Type ;
249
250 CastName
251   ::= "trunc" | "sext" | "zext"
252      | "ptrcast"
253      | "itof" | "ftoi" ;

```

```

254 ; -----
255 ; Values and temporaries
256 ; -----
257
258
259 Temp
260   ::= "t" Int ;
261
262 Value
263   ::= Temp | Const ;
264
265 ; -----
266 ; Constants
267 ; -----
268
269 ; Const can be scalar (numbers, chars, null) or aggregate (arrays).
270 Const
271   ::= ScalarConst | AggregateConst ;
272
273 ScalarConst
274   ::= "#" Int ":" Type           ; integer literal with explicit type
275   |   "#" CharLit ":" "char"     ; character literal (1 byte)
276   |   "#" FloatLit ":" "float"    ; float literal
277   |   "null" ":" Type ;          ; typed null (typically ptr<...>)
278
279 ; Aggregate constants currently model array initializers.
280 AggregateConst
281   ::= ArrayConst ;
282
283 ; Array constant:
284 ;   { <c0>, <c1>, ... } : array<T,N>
285 ; Element count should match N (or be validated by the compiler if you allow padding).
286 ArrayConst
287   ::= "{" [ Const { "," Const } ] "}" ":" ArrayType ;
288
289 ArrayType
290   ::= "array" "<" Type "," Int ">" ;
291
292 ; -----
293 ; Types
294 ; -----
295
296 ; Fully explicit types, including void and parametric types.
297 Type
298   ::= "void"
299   |   "int32" | "char" | "uchar" | "float" | "bool"
300   |   "ptr" "<" Type ">"
301   |   "array" "<" Type "," Int ">"
302   |   "struct" Ident ;
303
304 ; -----
305 ; Lexical rules (simplified)
306 ; -----

```



```
307
308 Ident
309   ::= Letter { Letter | Digit | "_" } ;
310
311 Int
312   ::= ["-"] Digit { Digit } ;    ; canonical IR integer format is decimal
313
314 FloatLit
315   ::= ["-"] Digit { Digit } "." Digit { Digit } ;
316
317 CharLit
318   ::= "'" ( Escape | AnyCharExceptQuote ) "'" ;
319
320 Escape
321   ::= "\n" | "\t" | "\\'" | "\\\\" ;
322
323 NL
324   ::= "\n" ;
325
326 Letter
327   ::= "A" | ... | "Z" | "a" | ... | "z" ;
328
329 Digit
330   ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
331
332 AnyCharExceptQuote
333   ::= ? any character except ' ? ;
```


Appendix E

Build and Usage Guide

Everything here is reproducible from a clean checkout on a recent macOS or Linux machine, and nothing requires a network connection once the prerequisites are installed. This appendix is the operational manual: how to build the compiler from source, how to drive it through each of its phases, how to run a compiled program, and how to build and test the companion project you write yourself.

E.1 Prerequisites

The compiler is a Java project built with Maven, and it runs its compiled output on a JavaScript simulator under Node.js. Three tools therefore need to be on the path, listed in Table E.1. A Java Development Kit at version 21 or later is required because the source uses language features introduced in Java 21; Maven 3.8 or later drives the multi-module build; and Node.js is needed only when you actually *run* a compiled program, because executing FRISC means handing the generated assembly to the simulator (Appendix C). Compiling, optimizing, and inspecting IR or assembly need only the JDK and Maven.

Tool	Version	Needed for
JDK	21 or later	building and running the compiler
Maven	3.8 or later	the multi-module build
Node.js	recent LTS or later	running compiled programs on the simulator

Table E.1: The three tools required to build the compiler and run its output. Node.js is needed only for the run step; the simulator itself ships with the repository.

E.2 Building the compiler

The FRISCcc compiler is an open-source repository [47]; clone it from <https://github.com/KarloKnezevic/ccompiler>. It ships a build script that wraps Maven. From the project root, `./build.sh` compiles every module and packages the command-line front end into a single runnable jar, skipping the test suite for speed; `./build.sh -t` runs the tests as well, and

`./build.sh -c` cleans previous build artifacts first. The script checks that a suitable Java is present and prints a clear diagnostic if it is not.

The build produces one jar containing the whole pipeline — lexer, parser, semantic analyzer, IR lowering, the optimization passes, the FRISC code generator, the IR interpreter, and the bytecode VM. The companion run script, described next, rebuilds this jar automatically whenever it notices that a source or specification file is newer than the jar, so in practice you rarely invoke the build script directly after the first time.

E.3 Driving the compiler

The run script, `./run.sh`, is the single entry point for exercising the compiler. It takes a flag selecting how far down the pipeline to go and a source file, and it writes each phase's artifact to an output directory. [Table E.2](#) lists the phase flags. Each flag implies the ones before it: asking for IR runs the lexer, parser, and semantic analyzer first, and asking for FRISC runs all of those plus the IR lowering and optimization. This is how you inspect any intermediate form — the token stream, the parse tree, the typed tree, the IR before and after optimization, the generated assembly — by stopping the pipeline where you want to look.

Invocation	Runs through
<code>./run.sh --lex prog.c</code>	tokenization
<code>./run.sh --parse prog.c</code>	+ parsing
<code>./run.sh --sem prog.c</code>	+ semantic analysis
<code>./run.sh --ir prog.c</code>	+ IR generation
<code>./run.sh --frisc prog.c</code>	+ FRISC code generation
<code>./run.sh --all --run prog.c</code>	full pipeline + simulation

Table E.2: The run script's phase flags. Each stage subsumes the earlier ones, so any flag lets you inspect every artifact produced up to that point.

Two further flags control optimization. `--00` disables the optimizer and emits code straight from lowering; `--01` enables the baseline optimization pipeline of [Chapter 7](#). Adding `--dump-ir` writes the IR both before and after optimization, which is the way to see what a pass actually changed. To compile a program with optimization and inspect the difference, for instance, you would run `./run.sh --ir --01 --dump-ir prog.c`; to compile and immediately run an optimized program, `./run.sh --all --run --01 prog.c`, whose printed number is the value the program returned ([Section C.3](#)).

Key insight

The most useful debugging flag is `--dump-ir`. Run `./run.sh --ir --01 --dump-ir prog.c` and the compiler writes the IR twice — once as lowering produced it and once after the optimizer ran. Reading the two side by side is the single fastest way to see

exactly what a pass changed, and it is how nearly every IR listing in this book was obtained.

E.4 Running IR directly: the interpreter and the VM

The two Part IV back ends run IR without going through FRISC code generation. The tree-walking interpreter of [Chapter 11](#) is invoked as `./run.sh run-ir file.ir`; adding `--trace-ir` prints a per-step trace of its execution. The bytecode VM of [Chapter 12](#) is invoked as `./run.sh run-vm file.ir`; it first lowers the IR to bytecode and then executes it. Adding `--trace-vm` prints a per-dispatch trace, and `--dump-bytecode` disassembles the lowered bytecode instead of running it, which is how the disassembly listings in [Chapter 12](#) were produced. Both back ends report the same result the simulator would, which is the property that lets the case studies of [Chapter 13](#) compare all three execution paths on the same programs.

E.5 Building and testing the companion

The book’s build-along experience lives in a separate open-source repository [\[46\]](#), the companion project, cloned from <https://github.com/KarloKnezevic/frisccc-companion>. In it you implement a compiler for *tinyC* — a deliberately smaller, integer-only C subset — one chapter at a time, with FRISCCc as the reference to check yourself against. The companion is a Maven multi-module project, one module per chapter, each laid out the same way: a `starter/` tree with stubbed functions marked for you to implement, a `solution/` tree with a working reference implementation, and a `tests/` tree of JUnit tests that verify the observable behaviour of whatever you build.

After cloning the companion, `mvn -q -DskipTests` package from its root builds every module, confirming the project compiles end to end. While working through a chapter you run just that chapter’s tests — for example `mvn test -pl ch03-lexer` for the tokenizer module — and iterate on the starter code until they pass; the solution tree is there to consult when you are stuck. The modules follow the chapter order ([Table E.3](#)), and each chapter’s “Your turn” section points at the specific starter files to edit. The first nine modules build the *tinyC* compiler itself, front end through bytecode VM; the last two (`ch13-casestudies` and `ch14-performance`) are measurement labs that exercise what you have built rather than extend it. Because *tinyC* compiles to the same FRISC target, the code you generate runs on the same simulator ([Appendix C](#)) the book uses for FRISCCc, so you can compare your output against the reference compiler’s directly.

Companion module	You build
ch03-lexer	the tinyC tokenizer
ch04-parser	the LR(1) parser
ch05- semantics	the type checker
ch06-ir	lowering to three-address IR
ch07-opt	optimization passes
ch08-codegen	the FRISC code generator
ch09-runtime	runtime helpers and frame layout
ch11-treewalk	a tree-walking IR interpreter
ch12-vm	a bytecode compiler and VM
ch13-casestudies	the three-way measurement lab
ch14-performance	the optimizer performance study

Table E.3: The chapter-aligned modules of the companion project. Each provides a starter, a reference solution, and a test suite; you fill in the starter until the tests pass.

Appendix F

Solutions

This appendix collects worked solutions to the Practice exercises at the end of each chapter. The Practice problems are the ones with definite answers — a trace to carry out, a derivation to complete, a number to compute, a rule to name — and every one of them is solved here. The chapter-end Projects are a different kind of work: open-ended extensions you carry out in the companion repository ([Appendix E](#)), and they have no single answer to print, so they are not reproduced here. Where a solution states a concrete artifact — an IR listing, an instruction count, a frame size — the number is the one the compiler actually produces; you can regenerate it yourself by running the program through the relevant phase ([Section E.3](#)).

F.1 Chapter 1: What we’re building, and why it’s worth a book

Exercise 1.1 (predicting the optimised IR for `return 5 + 3 * 2`;). The expression `5 + 3 * 2` has two binary operations: a multiplication with two constant operands (`3 * 2`) and an addition whose right operand is the result of that multiplication. Before the optimiser runs, the IR builder lowers them bottom-up, generating three instructions in block `L0`:

```
t0 = mul #3:int32, #2:int32 : int32
t1 = add #5:int32, t0 : int32
ret t1
```

The `TypedConstantFoldingPass` inspects each `BinOp` instruction and asks whether *both* operands are compile-time constants (`IrConst` values in the IR’s internal model). For the `mul` instruction both operands are integer literals, so the pass replaces the whole right-hand side with the precomputed constant `#6:int32`. After this transformation the first instruction becomes a constant-copy assignment:

```
t0 = #6:int32
```

The add instruction, however, still refers to the *temporary* `t0`, not to an `IrConst` directly. Because the constant-folding pass only folds instructions whose operands are both literal constants, it does not touch the add. The pass for propagating known values through memory slots (`GlobalValuePropagationPass`) operates on named slots in the stack frame, not on IR temporaries, so it does not substitute `t0 → #6` either. The pipeline runs until no pass makes a further change; after the first sweep, no instruction changes again.

The actual optimised IR that `./run.sh --frisc --01 --dump-ir` writes to `compiler-bin/ir-dumps/<program>/after_optimization.ir` is therefore (note the dump only appears for a stage that actually runs the optimiser, such as `--frisc --01 --dump-ir`; a bare `--ir` stops before the pipeline and emits no post-optimisation file):

```
L0:
    t0 = #6:int32
    t1 = add #5:int32, t0 : int32
    ret t1
```

Three instructions remain (the same count as before optimisation, though with the `mul` replaced), and the final result is still eleven. A reader who predicted a single `ret #11:int32` was reasoning correctly about what an idealised constant-propagation pass *could* do; the `FRISCcc` pass applies a more conservative rule, and the remaining add survives into the generated assembly. The code generator faithfully emits a `MOVE 6, R0` for `t0`, a `PUSH/ADD/POP` sequence for the add, and the simulator still prints 11.

Exercise 1.2 (which instruction changed, and which pass changed it). Place the two listings side by side:

Unoptimised IR (Listing 1.6)	Optimised IR (Listing 1.7)
<code>t0 = mul #5:int32, #3:int32 : int32</code>	<code>t0 = #15:int32</code>
<code>ret t0</code>	<code>ret t0</code>

The only changed instruction is the first one. In the unoptimised program it is an arithmetic `mul` instruction with two constant operands; in the optimised program it has been replaced by a direct constant assignment, `t0 = #15:int32`, with the multiplication computed at compile time and the result baked in as a literal. The `ret` line is textually identical in both listings.

The pass responsible is **TypedConstantFolding** (listed in Chapter 7 as `TypedConstantFoldingPass`). It is the only pass in the pipeline that rewrites an arithmetic instruction into a constant when both operands are compile-time-known values. The name is exact: the folding is *typed* because the pass preserves the `int32` annotation on the result literal, not just the numeric value. No other pass in the pipeline would have found this pattern; the remaining sixteen passes look for other kinds of redundancy (unreachable blocks, dead temporaries, repeated subexpressions, loop-invariant code, and so on) and have nothing to rewrite in a two-instruction function with no control flow.

Exercise 1.3 (HALT and the fetch-decode-execute loop). The HALT instruction appears on line 11 of Listing 1.8, inside the program entry-point block at the *very beginning* of the generated file, not at the end. The entry-point block occupies lines 9–11:

```
MOVE 40000, R7    ; Initialize stack pointer
CALL F_MAIN       ; Call main
HALT              ; Program end
```

Everything after line 11 is function definitions — F_MAIN and the helper routines. Those definitions are data in the assembly file, not a continuation of the control-flow path. The processor reaches HALT by returning from F_MAIN through the RET instruction inside the function epilogue; that RET pops the return address pushed by CALL F_MAIN and jumps back to the instruction immediately following the CALL, which is precisely HALT.

When the friscjs simulator’s fetch-decode-execute loop encounters HALT, it reads the final value of R6 (the return-value register), prints it, and terminates the simulation loop without fetching another instruction. If HALT were never reached — for example, because the function entered an infinite loop — the simulator would continue fetching and executing instructions indefinitely. In a flat address space with no operating system or timer interrupt, there is no external mechanism to stop it; the program would run until the host process was killed or the simulated instruction counter overflowed.

F.2 Chapter 2: The shape of a compiler

Exercise 2.1 (the case for and against a merged scanner-parser). *The case for merging.* A single scanner-parser phase eliminates the boundary artifact entirely: the implementation no longer needs to define, allocate, or pass a token stream between two phases, which reduces memory traffic and removes one contract boundary that has to be tested and maintained. Some early Pascal compilers exploited this by running a hand-written recursive-descent parser whose terminal procedures consumed characters directly, achieving a remarkably small working set. When the target grammar is simple and the token vocabulary is small, the single-pass structure is easier to explain and easier to modify.

The case against. The grammar and the lexical rules are governed by different mathematical formalisms — regular languages for the lexer, context-free grammars for the parser — and mixing them in one pass tangles two very different kinds of reasoning. Adding or changing a token kind (say, introducing a new keyword) requires editing the parser’s character-consumption logic; adding or changing a syntactic construct requires editing the same code that handles lexical decisions. The combined phase is harder to test in isolation, harder to replace independently, and harder to reason about formally. Any diagnostic the combined phase emits must carry the context of both phases, making error messages either vague or disproportionately complex.

FRISCCc’s choice. FRISCCc keeps the phases separate. The primary reason is the one given in the second argument above: the artifact contract between a lexer and a parser is a

clean interface that lets the two phases be developed, tested, and replaced independently. As the chapter notes in [Section 2.1](#), the parser never receives a malformed token because the lexer’s contract forbids them; collapsing the two phases would dissolve that guarantee without any compensating benefit at FRISCcc’s scale.

Exercise 2.2 (FRISC stack depth and overflow behaviour). The stack pointer R7 is initialised to address 40000 and grows *downward* on each PUSH or frame-allocation SUB R7, N, R7. With the heap at address 0 growing upward and static data ending at address 1000, the last address the stack can safely occupy without colliding with static data is 1001 (the first address above the static data region). The available stack window is therefore $40000 - 1001 = 38999$ bytes, which accommodates just under $\lfloor 38999/4 \rfloor = 9749$ 32-bit words of stack content at one time. In practice each active stack frame in FRISCcc consumes at minimum 8 bytes (the saved old frame pointer and the function’s return address), so a purely recursive call chain can be no deeper than roughly 4875 frames before the pointer crosses the 1000 boundary.

If a recursive function pushes enough frames to drive R7 below address 1001, the FRISC hardware has no mechanism to detect the overflow. There is no stack-limit register, no trap, no memory-protection unit, and no operating system to catch the fault. The simulator sees R7 decrease past 1000 and simply continues: subsequent STORE instructions through the frame pointer will write into the static-data region (addresses 0 through 999), silently corrupting whatever global or constant data lives there. The compiled program will then produce incorrect results or loop forever, with no indication at run time that anything has gone wrong.

Exercise 2.3 (three-address code, SSA, and FRISCcc’s choice). Section 3.2 of Lattner and Adve [53] describes LLVM’s IR as a three-address code in static single-assignment (SSA) form.

What property of three-address code does SSA preserve? SSA retains the defining characteristic of three-address code: every instruction names an explicit destination and at most two explicitly named source operands, making the dataflow between instructions visible in the instruction text rather than implicit in a stack discipline.

What property does SSA add? SSA adds the invariant that each name is defined by *exactly one instruction* in the entire function. At control-flow join points where two execution paths each define the same source-level variable, SSA inserts a ϕ -function whose job is to select the appropriate incoming definition based on which predecessor block was executed. The uniqueness-of-definition invariant means that the sole definition of any name is trivially locatable, which makes every def-use chain a pointer to a unique site rather than a search through a set of possible definitions.

What does FRISCcc give up, and what does it gain? By staying with plain (non-SSA) three-address code, FRISCcc gives up the trivial def-use chains that SSA provides: when a name is used in an instruction, finding its definition requires a dataflow analysis rather than a direct lookup. Several of the passes in Chapter 7 therefore carry explicit lattice-based

analyses (*reaching definitions*, *available expressions*) that would be nearly trivial under SSA. What FRISCCc gains is the avoidance of the SSA construction algorithm itself — computing dominance frontiers, placing ϕ -functions, and renaming definitions is a non-trivial pass that would consume a chapter’s worth of explanation and implementation work. As the sidebar in [Section 2.2](#) notes, the sixteen passes in FRISCCc are individually small enough to fit in the reader’s head without ϕ -functions, and the pedagogical gain of showing real analyses at a slightly higher cost is worth the loss of the SSA shortcut.

F.3 Chapter 3: Reading characters: tokenization

Exercise 3.1 (Subset construction — two rows). The Thompson NFA for $a(b|c)^*$ built by the chapter’s inductive gadgets has eight states. We label the start state q_0 and the unique accepting state q_7 . Following the epsilon edges precisely:

- q_0 : the concatenation gadget’s outer start. One character transition on a leads to q_1 . Two epsilon edges from q_0 lead directly into the Kleene-star wrapper: $q_0 \xrightarrow{\varepsilon} q_2$ (the loop-entry state, whose job is to fan out to the alternation body or bypass it) and $q_0 \xrightarrow{\varepsilon} q_7$ (the zero-repetition bypass, reaching the overall accept state).
- From q_2 two epsilon edges fan out to q_3 (for b) and q_5 (for c).
- q_4 is the accept of the b branch; q_6 is the accept of the c branch. Both have epsilon edges back to q_2 (the back-edge that enables further repetitions) and an epsilon edge forward to q_7 (so that one or more repetitions can accept).

Taking the epsilon-closure of the NFA start $\{q_0\}$ therefore reaches: q_0 itself; then q_2 via the loop-entry epsilon; then q_3 and q_5 via the alternation fan-out; then q_7 via the bypass epsilon. The states q_4 and q_6 are not reachable from q_0 without first consuming a character, so they do not enter the initial closure.

$$D_0 = \varepsilon\text{-cl}(\{q_0\}) = \{q_0, q_2, q_3, q_5, q_7\}$$

Because $q_7 \in D_0$ the DFA state D_0 is **accepting**: the empty string matches $a(b|c)^*$ (zero repetitions of the Kleene body and the literal a has not been consumed yet — but note that the NFA’s accept here is the final accept of the overall machine, so the empty string is *not* actually in the language; the exercise asks which DFA states are accepting in the technical sense, and D_0 is technically accepting because it contains the NFA’s final state. The language that $a(b|c)^*$ denotes does not contain ε : the character a is mandatory. The apparent contradiction resolves when we remember that an accepting DFA state means the machine *can* accept from this point if the input ends here, not that the scan so far has consumed a valid string from the start. In maximal munch, D_0 is the DFA start before any character has been consumed, so no token is committed yet regardless of the accepting flag.)

From D_0 , on character **a**: the only state in D_0 that has an outgoing transition on **a** is q_0 (via the literal **a** edge to q_1). All other states in D_0 — q_2, q_3, q_5, q_7 — have no **a** transition. So we collect $\{q_1\}$ and take its epsilon-closure. From q_1 (the concatenation midpoint) an epsilon edge leads into the same Kleene-star structure: $q_1 \xrightarrow{\varepsilon} q_2$, from which the alternation fan-out and bypass proceed exactly as before. The closure is:

$$D_1 = \varepsilon\text{-cl}(\{q_1\}) = \{q_1, q_2, q_3, q_5, q_7\}$$

Again $q_7 \in D_1$, so D_1 is also an **accepting** DFA state — which is correct, because the string **a** alone (with zero repetitions of the Kleene body) is in the language. The two worklist rows are:

DFA state	NFA subset	Accepting?	On a	On b/c
D_0	$\{q_0, q_2, q_3, q_5, q_7\}$	yes	D_1	(further states)
D_1	$\{q_1, q_2, q_3, q_5, q_7\}$	yes	(dead)	(further states)

Note that $D_0 \neq D_1$ because $q_0 \in D_0$ but $q_0 \notin D_1$, while $q_1 \in D_1$ but $q_1 \notin D_0$; the worklist algorithm must therefore track both as distinct DFA states. Completing the construction would add one more state for the b/c branches (both lead to the same NFA subset $\{q_4 \text{ or } q_6, q_2, q_3, q_5, q_7\}$ after epsilon-closure, which collapses to a single DFA state D_2 accepting on both **b** and **c**) and a dead (non-accepting) sink for all other characters.

Exercise 3.2 (Mode-machine trace for $x = /* \text{ hi } */ \emptyset;$). The fragment $x = /* \text{ hi } */ \emptyset;$ is eighteen bytes (counting the space before $\emptyset$ and the closing semicolon). We trace the lexer boundary by boundary. The four mode names, taken from `config/lexer_definition.txt`, are: `S_pocetno` (initial), `S_komentar` (block-comment), `S_jednolinijskiKomentar` (line-comment), and `S_string` (string).

Segment	Bytes	Mode in	Match	Token emitted?	Mode out
x	1	<code>S_pocetno</code>	x	yes: <code>IDN(x)</code>	<code>S_pocetno</code>
	1	<code>S_pocetno</code>		no (whitespace)	<code>S_pocetno</code>
=	1	<code>S_pocetno</code>	=	yes: <code>OP_PRIDRUZI</code>	<code>S_pocetno</code>
	1	<code>S_pocetno</code>		no (whitespace)	<code>S_pocetno</code>
/*	2	<code>S_pocetno</code>	/*	no	<code>S_komentar</code>
hi	4	<code>S_komentar</code>	each byte	no (discarded)	<code>S_komentar</code>
*/	2	<code>S_komentar</code>	*/	no	<code>S_pocetno</code>
	1	<code>S_pocetno</code>		no (whitespace)	<code>S_pocetno</code>
∅	1	<code>S_pocetno</code>	∅	yes: <code>BROJ(∅)</code>	<code>S_pocetno</code>
;	1	<code>S_pocetno</code>	;	yes: <code>TOCKAZAREZ</code>	<code>S_pocetno</code>

The complete token stream handed to the parser is:

`IDN(x), OP_PRIDRUZI, BROJ(∅), TOCKAZAREZ`

Four tokens. The block comment’s two delimiter pairs and four-character body consume ten bytes and produce nothing. Notice that inside `S_komentar` the characters `h`, `i`, and the surrounding spaces are each matched individually by the catch-all rule `<S_komentar>{sviZnakovi}` and silently discarded; the mode returns to `S_pocetno` only when the two-byte sequence `*/` is consumed by the dedicated rule `<S_komentar>\*/`.

Exercise 3.3 (Maximal munch on `inta = 5;`). When the lexer scans `inta = 5;` it begins in `S_pocetno`. The very first character is `i`. The identifier DFA and several keyword DFAs (for `int`, `if`) advance in parallel through the merged machine. After consuming `i`, `n`, `t` the keyword rule for `KR_INT` is in an *accepting* state — but the identifier rule is also still alive, because the identifier regex `(_|{znak})(_|{znak}|{znamenka})*` can continue as long as the next character is a letter, digit, or underscore.

The lexer does *not* emit `KR_INT` at this point. Maximal munch means the lexer records position 3 (just past the `t`) as its last-accept snapshot but continues driving the DFA forward. On character `a` (position 4) the keyword sub-machine for `int` has no outgoing transition — `int` does not extend to `inta` as a keyword — but the identifier sub-machine does, because `a` is a valid identifier continuation. So the DFA advances to a new state that is still alive. After consuming `a` the DFA again reaches an accepting state (identifier), updating the last-accept snapshot to position 4. The next character is a space, at which neither the identifier nor any keyword can extend; the DFA dies, and the lexer commits to the snapshot at position 4.

The first token is therefore `IDN(inta)`: a four-character identifier. The keyword rule for `int` was a candidate, but maximal munch chose the longer match. Because the longer match is produced by the identifier rule (not the keyword rule), the final token kind is `IDN`, not `KR_INT`. The complete token stream for `inta = 5;` is:

```
IDN(inta),  OP_PRIDRUZI,  BROJ(5),  TOCKAZAREZ
```

The commit point is key: the lexer never emits a partial match when the DFA is still alive. “Accepting state” in the DFA means “a valid token ends here if the input ends here or the DFA dies next”; it does not mean “emit immediately”. The maximal-munch loop holds the last-accept position in reserve and fires only when no further extension is possible.

Exercise 3.4 (Regex for `OP_POW`, rule order, and two inputs). The regex for `OP_POW` is simply:

```
\*\*
```

That is, two literal asterisks in sequence. In the specification file, an asterisk must be escaped with a backslash because it is a regex metacharacter.

Rule order. The new `OP_POW` rule must appear *before* the existing `ASTERISK` rule in `config/lexer_definition.txt`. The reason is the tie-break half of the maximal-munch policy: when two rules produce a match of the same length, the rule that appears earlier in the specification

wins. On the input `**`, the `ASTERISK` rule matches a single `*` with length 1, while `OP_POW` matches `**` with length 2. So on `**`, maximal munch already prefers `OP_POW` on length grounds alone — the `OP_POW` match (2 characters) is longer than the `ASTERISK` match (1 character).

However, the rule-order question is not vacuous. Suppose someone wrote a rule for `OP_POW_ALT` that also matched `**` (say, by a different regex path). Both would match length 2 and the tie-break would fire. In the actual specification with exactly these two rules, there is no tie on length, so rule order is not strictly necessary — but it is good practice to place longer patterns first, because rule order is the only protection against future ambiguity. If `ASTERISK` were placed before `OP_POW` *and* the DFA accepted length-1 matches eagerly without looking further, the `**` input would be tokenised as two separate `ASTERISK` tokens. Maximal munch prevents this: the DFA advances to the second `*` and sees a longer accepting match for `OP_POW`, so `OP_POW` wins regardless. But placing `OP_POW` first is the conventional, defensive approach.

Token streams for the two inputs.

- `2 * *3` (with a space between the stars): the space between the two `*` characters prevents the merged DFA from ever reaching the `OP_POW`-accepting state. After scanning `2` the lexer emits `BROJ(2)`; then is whitespace (discarded); then `*` is a single-character match for `ASTERISK`; then is discarded; then `3` is `BROJ(3)`. The stream is:

`BROJ(2),    ASTERISK,    BROJ(3)`

- `2**3` (no space): after emitting `BROJ(2)`, the lexer begins scanning at the first `*`. The merged DFA advances through both `*` characters; after the second `*` it reaches the accepting state for `OP_POW` (length 2). The next character is `3`, which the `OP_POW` DFA cannot extend, so the DFA dies. The last-accept snapshot is length 2 with kind `OP_POW`; the lexer emits `OP_POW` and then emits `BROJ(3)`. The stream is:

`BROJ(2),    OP_POW,    BROJ(3)`

The two inputs differ because the space in `2 * *3` causes the DFA to die after one `*`, committing to `ASTERISK` before the second `*` is even considered. In `2**3` the DFA never dies mid-way: both asterisks are consumed in one uninterrupted run of the maximal-munch loop.

Exercise 3.5 (Pumping lemma for nested block comments). We want to show that the language L_{nest} of well-nested C-style block comments — strings of the form `/*w*/` where *w* may itself contain nested `/*...*/` pairs at arbitrary depth — is not regular.

Suppose for contradiction that L_{nest} is regular. Then the pumping lemma gives us a pumping length $p \geq 1$ such that every string in L_{nest} with length $\geq p$ can be written as xyz with $|xy| \leq p$, $|y| \geq 1$, and $xy^iz \in L_{\text{nest}}$ for all $i \geq 0$. Choose the string

$$s = \begin{array}{c} /* \dots /* \quad */ \dots */ \\ |\blacksquare\{Z\blacksquare\} \quad |\blacksquare\{Z\blacksquare\} \\ p+1 \text{ opens } p+1 \text{ closes} \end{array}$$

which has $p + 1$ opening delimiters followed by $p + 1$ closing delimiters, with each delimiter being the two-character sequence `/*` or `*/`. This string is in L_{nest} (the outermost pair encloses p nested layers), and its length is $4(p + 1) \geq p$. Because $|xy| \leq p$ and each opening delimiter is two characters, the substring y lies entirely within the first p characters of s , which are entirely composed of characters from the opening delimiters (`/` and `*`, in that pattern). Since $|y| \geq 1$ the string y consists of one or more characters drawn from those opening delimiters only. Pumping down to $i = 0$ removes $|y| \geq 1$ characters from the opening portion, reducing the number of complete `/*` pairs in the prefix while leaving all $p + 1$ closing `*/` pairs intact. The result has strictly fewer opens than closes, so it cannot be a well-nested sequence of block comments at any depth. It is not in L_{nest} .

This is a contradiction. Therefore L_{nest} is not regular.

The engineering consequence is direct: recognising nested block comments requires counting (keeping a depth counter), and counting requires unbounded memory. A finite-state machine has no unbounded memory, so no DFA can recognise L_{nest} . C resolves this by defining `*/` to *always* close the innermost open comment, regardless of depth, turning the language of comments back into the regular language handled by FRISCCc's block-comment mode. Languages that do allow nested comments — ML, Haskell, D — must handle them with a counter in the lexer, stepping outside the pure regular fragment.

F.4 Chapter 4: Reading structure: parsing

Exercise 4.1 (First sets by fixpoint iteration). The grammar has three non-terminals and six productions, reproduced for reference:

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T \cdot F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow \text{num}$

The FIRST rules are: (a) every terminal at the beginning of a right-hand side adds itself to the FIRST set of the non-terminal on the left; (b) if a right-hand side begins with a non-terminal X , every element of $\text{First}(X)$ contributes to the FIRST set of the left-hand non-terminal; (c) epsilon contributions propagate similarly, but this grammar has no epsilon productions.

Round 0 (initial): all sets empty.

$$\text{First}(E) = \{\}, \quad \text{First}(T) = \{\}, \quad \text{First}(F) = \{\}$$

Round 1: scan each production.

- Production (1): $E \rightarrow E + T$. The right-hand side begins with E . $\text{First}(E)$ is currently empty, so nothing is added to $\text{First}(E)$ yet.
- Production (2): $E \rightarrow T$. Right-hand side begins with T . $\text{First}(T)$ is currently empty; nothing added.
- Production (3): $T \rightarrow T \cdot F$. Right-hand side begins with T . Nothing added yet.
- Production (4): $T \rightarrow F$. Right-hand side begins with F . $\text{First}(F)$ is empty; nothing added.
- Production (5): $F \rightarrow (E)$. Right-hand side begins with the terminal $($. Add $($ to $\text{First}(F)$.
- Production (6): $F \rightarrow \text{num}$. Right-hand side begins with the terminal num . Add num to $\text{First}(F)$.

After Round 1:

$$\text{First}(E) = \{\}, \quad \text{First}(T) = \{\}, \quad \text{First}(F) = \{ (, \text{num} \}$$

Round 2: re-scan.

- Productions (1) and (3): still begin with E and T respectively, which are still empty. Nothing new from these.
- Production (2): $E \rightarrow T$. $\text{First}(T)$ is empty; nothing added yet.
- Production (4): $T \rightarrow F$. $\text{First}(F) = \{ (, \text{num} \}$. Add $($ and num to $\text{First}(T)$.
- Production (5) and (6): F 's own productions; $\text{First}(F)$ unchanged.

After Round 2:

$$\text{First}(E) = \{\}, \quad \text{First}(T) = \{ (, \text{num} \}, \quad \text{First}(F) = \{ (, \text{num} \}$$

Round 3: re-scan.

- Production (2): $E \rightarrow T$. $\text{First}(T) = \{ (, \text{num} \}$. Add $($ and num to $\text{First}(E)$.
- Production (3): $T \rightarrow T \cdot F$. $\text{First}(T) = \{ (, \text{num} \}$. The set $\text{First}(T)$ gains nothing new (it already contains those values).

- Production (1): $E \rightarrow E + T$. $\text{First}(E) = \{ (, num \}$ after this round's update; nothing beyond what production (2) provided.
- All other productions: no changes.

After Round 3:

$$\text{First}(E) = \{ (, num \}, \quad \text{First}(T) = \{ (, num \}, \quad \text{First}(F) = \{ (, num \}$$

Round 4 (convergence check): re-scan all productions. No production can contribute a new terminal to any set: every set already contains everything reachable in one step from the terminals $($ and num . The fixpoint has been reached.

All three non-terminals have the same FIRST set, $\{ (, num \}$, which makes intuitive sense: every arithmetic expression — whether it is an E , a T , or an F — must begin with either a left parenthesis or a number. No second round was needed after Round 3 because the only source of new terminals was $\text{First}(F)$, which was fully populated in Round 1 from the two base productions. Rounds 2 and 3 merely propagated that same set up through T and E respectively. Once every non-terminal points at $\{ (, num \}$ there is nowhere left for any new terminal to appear: the alphabet of the grammar has only five symbols $(, +, \cdot,), num$, and both operators can never begin a production's right-hand side without a non-terminal appearing first.

Exercise 4.2 (*goto(I_0, T) by hand*). We work from I_0 as constructed in Section 4.5 of the chapter. Recall that I_0 is the closure of the single seed item $[S' \rightarrow \cdot E, \$]$. Taking the closure adds items for every production whose left-hand side appears after a dot; for the three-rule grammar the full set I_0 contains six items:

$$\begin{aligned} &[S' \rightarrow \cdot E, \$] \\ &[E \rightarrow \cdot E + T, \$ / +] \\ &[E \rightarrow \cdot T, \$ / +] \\ &[T \rightarrow \cdot T \cdot F, \$ / + / \cdot] \\ &[T \rightarrow \cdot F, \$ / + / \cdot] \\ &[F \rightarrow \cdot (E), \$ / + / \cdot] \\ &[F \rightarrow \cdot num, \$ / + / \cdot] \end{aligned}$$

To compute $\text{goto}(I_0, T)$ we collect every item in I_0 whose dot sits immediately before the non-terminal T , advance each dot, and take the closure of the resulting kernel. Two items in I_0 qualify:

$$[E \rightarrow \cdot T, \$ / +] \quad \text{and} \quad [T \rightarrow \cdot T \cdot F, \$ / + / \cdot]$$

Advancing the dots gives the kernel:

$$\{[E \rightarrow T \cdot, \$ / +], [T \rightarrow T \cdot \cdot F, \$ / + / \cdot]\}$$

Taking the closure: neither item's dot precedes a non-terminal (the dot in $E \rightarrow T \cdot$ is at the rightmost position; the dot in $T \rightarrow T \cdot \cdot F$ precedes the terminal $\cdot$), so the closure adds nothing. The result is I_2 as named in the chapter, with exactly two items.

The lookahead sets come from propagation through the original I_0 items. In $E \rightarrow T \cdot$ the lookahead is $\{+, \$\}$ (the same lookahead that the item $[E \rightarrow \cdot T, \$/+]$ carried in I_0). In $T \rightarrow T \cdot \cdot F$ the lookahead is $\{+, \cdot, \$\}$ (propagated from $[T \rightarrow \cdot T \cdot F, \$/+/\cdot]$). Any difference between your lookaheads and the chapter's description is most likely a sign that you carried over the lookaheads for $E \rightarrow T$ (which follows E , hence has lookahead $+$ and $\$$) rather than those for $T \rightarrow T \cdot F$ (which follows T , hence also admits $\cdot$). The items with dot-past-a-non-terminal retain exactly the lookaheads of the items they were advanced from.

Exercise 4.3 (Tracing $(a + b) * c$). We show every step using the ACTION/GOTO table of Table 4.1. The stack grows leftward and the parser begins in state 0. Each a , b , c is treated as *num*, state 5 on shift; $($ shifts to state 4; we use the production numbering from the table's legend: (1) $E \rightarrow E + T$, (2) $E \rightarrow T$, (3) $T \rightarrow T \cdot F$, (4) $T \rightarrow F$, (5) $F \rightarrow (E)$, (6) $F \rightarrow \text{num}$.

step	stack	remaining input	action	note
1	0	(a + b) * c \$	shift 4	read (
2	4 0	a + b) * c \$	shift 5	read a
3	5 4 0	+ b) * c \$	reduce 6	$F \rightarrow \text{num}$
4	3 4 0	+ b) * c \$	reduce 4	$T \rightarrow F$
5	2 4 0	+ b) * c \$	reduce 2	$E \rightarrow T$
6	8 4 0	+ b) * c \$	shift 6	read +
7	6 8 4 0	b) * c \$	shift 5	read b
8	5 6 8 4 0	) * c \$	reduce 6	$F \rightarrow \text{num}$
9	3 6 8 4 0	) * c \$	reduce 4	$T \rightarrow F$
10	9 6 8 4 0	) * c \$	reduce 1	$E \rightarrow E + T$
11	8 4 0	) * c \$	shift 11	read)
12	11 8 4 0	* c \$	reduce 5	$F \rightarrow (E)$
13	3 0	* c \$	reduce 4	$T \rightarrow F$
14	2 0	* c \$	shift 7	read *
15	7 2 0	c \$	shift 5	read c
16	5 7 2 0	\$	reduce 6	$F \rightarrow \text{num}$
17	10 7 2 0	\$	reduce 3	$T \rightarrow T \cdot F$
18	2 0	\$	reduce 2	$E \rightarrow T$
19	1 0	\$	accept	tree complete

(States 8, 9, 10, and 11 are the parenthesis-handling states; all four appear as their own rows in Table 4.1, so every action above can be checked against the chapter's table directly.)

The resulting parse tree has the multiplication $*$ at the root, with $F \rightarrow (E)$ as its left child and the identifier c as its right. Inside the F node, the E subtree is rooted at the addition

+, with a and b as its leaves. Compare this to $a + b * c$: in that tree the addition is at the root and the multiplication is the right child of the addition. The one node that changes is the root: the parentheses wrap $a + b$ into a single F node, which sits at the bottom of the precedence cascade, so the following multiplication must sit *above* it, reversing the tree's shape.

Exercise 4.4 (Dangling else: two trees and the conflict). Consider the fragment `if (e1) if (e2) S1 else S2`. The two parses correspond to the two ways the `else` can attach.

Outer-if binding (non-standard):

$$\begin{aligned} \langle \text{if-stmt} \rangle &\rightarrow \text{if } (e1) \langle \text{if-stmt} \rangle \text{ else } S2 \\ \text{where } \langle \text{if-stmt} \rangle &\rightarrow \text{if } (e2) S1 \end{aligned}$$

The outer `if` takes the `else` as its alternate branch; the inner `if` has no `else`. The two subtrees share the boundary after $S1$: the outer tree's then arm ends at $S1$ and the `else` belongs to the outer `if`.

Inner-if binding (C standard):

$$\begin{aligned} \langle \text{if-stmt} \rangle &\rightarrow \text{if } (e1) \langle \text{if-stmt} \rangle \\ \text{where } \langle \text{if-stmt} \rangle &\rightarrow \text{if } (e2) S1 \text{ else } S2 \end{aligned}$$

The inner `if` takes the `else`; the outer `if` has no `else`. The inner then branch is $S1$; the inner `else` branch is $S2$; and the outer `if`'s body is the entire inner `if-else` statement.

The conflict. The parser reduces $S1$ as the inner `if`'s then-body. At that point the item set contains a complete item $[\langle \text{if-stmt} \rangle \rightarrow \text{KR_IF } \dots \langle \text{stmt} \rangle \cdot, \text{KR_ELSE}]$ (ready to reduce the inner `if` without an `else` on lookahead `KR\_ELSE`) and a partial item $[\langle \text{if-stmt} \rangle \rightarrow \text{KR_IF } \dots \langle \text{stmt} \rangle \cdot \text{KR_ELSE } \langle \text{stmt} \rangle, a]$ (ready to shift `KR\_ELSE` and extend toward the two-arm form). This is a shift-reduce conflict on lookahead `KR\_ELSE`.

Why “shift wins” selects the inner binding. When the table-builder resolves the conflict in favour of shift, the parser shifts the `else` token and continues building the longer production $\langle \text{if-stmt} \rangle \rightarrow \dots \langle \text{stmt} \rangle \text{KR_ELSE } \langle \text{stmt} \rangle$. This attaches the `else` to the production that is currently on top of the stack — which is the inner `if` — so the inner `if` acquires the `else`. Had the parser reduced instead, it would have closed the inner `if` without an `else` and left the `KR\_ELSE` token for the outer `if` to consume; that is the outer binding. The “shift wins” convention is therefore not arbitrary: it encodes the “nearest `if`” rule that every C programmer expects, and it does so without any changes to the grammar.

Exercise 4.5 (Right-recursive associativity). The right-recursive production $\langle \text{add} \rangle \rightarrow \langle \text{mul} \rangle \mid \langle \text{mul} \rangle + \langle \text{add} \rangle$ still generates every additive expression because it is merely a re-parameterisation: every string of the form $m_1 + m_2 + \dots + m_k$ (where each m_i is a multiplicative expression) is derivable by applying the second production $k-1$ times, consuming one $\langle \text{mul} \rangle$ factor at each step from the left, with the recursion resolving the tail.

The difference is *how the tree groups the factors*. The left-recursive rule builds the tree left-to-right, associating early: $((m_1 + m_2) + m_3)$. The right-recursive rule groups the tail first and associates late: $(m_1 + (m_2 + m_3))$.

For the specific input $a - b - c$ with $a = 4, b = 3, c = 1$: under *left* recursion the tree is $((a - b) - c)$, giving $(4 - 3) - 1 = 0$, which is the C programmer's expectation. Under *right* recursion the tree is $(a - (b - c))$, giving $4 - (3 - 1) = 2$, a different numeric result. The right-recursive grammar is thus incorrect for subtraction and division: it generates the right set of *strings* but imposes the wrong *meaning*. This is precisely why the FRISCCc grammar (and C's grammar) uses left-recursive productions for left-associative operators.

F.5 Chapter 5: Meaning, not just shape

Exercise 5.1 (Definitely-returns and while (1)). The definitely-returns predicate $DR(s)$ for a statement s is defined structurally on the statement's kind.

- $DR(\text{return } e) = \text{true}$.
- For any declaration, expression statement, or `break/continue`: $DR(s) = \text{false}$.
- For a compound statement $\{s_1, s_2, \dots, s_n\}$: $DR(\{s_i\}) = s_1 \vee s_2 \vee \dots \vee s_n$, where we set the flag to *true* as soon as any constituent statement returns. (Statements after the first returning one are unreachable, but the predicate does not diagnose that here; dead-code warnings are a separate pass.)
- For `if (e) s1 else s2`: $DR(\text{if/else}) = DR(s_1) \wedge DR(s_2)$. Both branches must definitely return, because only one executes at run time: if either branch falls through, the whole construct falls through.
- For `while (e) s0` (and analogously for `for`): $DR(\text{while}) = \text{false}$.

The body s_0 may individually be a definitely-returning statement, but the loop as a whole does not definitely return. The reason is conservative: if the guard e evaluates to false on the first iteration, the body never executes and the program falls through to whatever follows the loop. The predicate therefore answers *false* regardless of what the body does.

Why does `while (1) { ... }` have to be treated conservatively? The guard is the literal 1, which a human reader can see is always non-zero. Proving that fact in general, however, requires knowing that the guard expression is a non-zero constant — or that the loop body

contains no break reachable without a prior return, which is itself a dataflow question. The semantic analyser of Chapter 5 does not perform dataflow; it is a single-pass post-order visitor. Deciding non-termination of arbitrary loops is undecidable in general (it reduces to the halting problem), so the analyser conservatively treats every loop as potentially non-executing and returns *false* from *DR*.

A program the conservative check rejects despite always returning at run time:

```

1  int main(void) {
2      while (1) {
3          return 42;
4      }
5  }
```

The body `return 42` definitely returns, but the loop wrapping it gets $DR = \text{false}$, so a *DR*-based check would flag the function body as potentially missing a return. At run time the loop body executes on every iteration, so the function can never fall off the end. The conservative rule is right in the sense of the specification but too pessimistic for this particular program.

It is worth being precise about what FRISCCc actually enforces here: *nothing*. FRISCCc does not implement the definitely-returns analysis at all — a non-void function that falls off the end without a return is accepted, and this program compiles cleanly. The predicate above is therefore the design for a stricter analyser, and adding it is precisely the exercise. Were such a check in place, the conservative rule would reject this always-returning program; adding an explicit `return 0;` after the loop body would silence the diagnostic without changing runtime behaviour.

Exercise 5.2 (Four programs the analyser rejects). Each of the four cases maps to a distinct analysis rule or auxiliary check.

(a) **Assignment to an rvalue.**

```

1  int main(void) { 5 = x; return 0; }
```

The assignment rule T-ASSIGN requires that the left-hand operand carry the *lvalue* attribute, meaning it must designate a memory location. The integer literal 5 is an rvalue — it produces a value but refers to no location — so the lvalue premise fails and the analyser emits “left-hand side of assignment must be an lvalue”.

(b) **Undeclared identifier.**

```

1  int main(void) { return z; }
```

The variable-lookup rule T-VAR performs $\Gamma(z)$ — a walk of the scope stack from top to bottom looking for a binding named z . No such binding exists in any frame, so the walk bottoms out and the analyser reports “undeclared identifier: z ”.

(c) Incompatible types in comparison.

```
1 | int main(void) { int i; float f; if (i < f) return 1; return 0; }
```

The comparison rule T-CMP requires both operands to share the same numeric kind, after applying the implicit-promotion relation $\prec$. Here $\text{int32} \prec \text{float}$ holds, so the analyser would normally promote i to float silently. In FRISCCc’s subset, however, the `commonNumeric` helper in `TypePromotion` is restricted: it accepts the promotion chain $\text{char} \prec \text{int32} \prec \text{float}$ upward but reports an incompatible-types diagnostic if the promoted common type is float in a context where floating-point comparison is not permitted by the currently analysed expression’s context. (A simpler failing case: use two structs or a pointer with an integer.) The diagnostic is “incompatible operand types in comparison”.

(d) break outside a loop.

```
1 | int main(void) { break; return 0; }
```

The loop-context auxiliary check in `JumpStatementRules` tracks a loop-depth counter, incremented on entry to any `while` or `for` node and decremented on exit. When the analyser visits the `break` node it checks the counter: zero means there is no enclosing loop, and the analyser emits “break outside loop”.

Exercise 5.3 (Transitivity of the promotion relation). The promotion relation is defined by two base rules, $\text{char} \prec \text{int32}$ and $\text{int32} \prec \text{float}$, together with the reflexivity axiom $\tau \prec \tau$ for every type τ . No explicit transitivity axiom is listed in the rule set of Section 5.4.

Is $\text{char} \prec \text{float}$ derivable? Yes, but only by chaining two applications of the base rules, not by a single rule. The relation as stated is not closed under transitivity by a single step; it becomes transitive only when you apply the rules two at a time. In practice FRISCCc’s `TypePromotion.commonNumeric` walks the upward chain $\text{char} \rightarrow \text{int32} \rightarrow \text{float}$ one step at a time, inserting a cast instruction at each step: first a `char`-to-`int` widening (zero-extension on FRISC), then an `int`-to-`float` lift via the runtime helper `F_I2F`. Transitivity therefore holds *constructively*: each step in the chain produces a real IR instruction rather than a single giant cast.

Does FRISCCc accept `char c = 'A'; float f = c;`? Yes. The assignment initialiser `c` has type `char`; the declared type of `f` is `float`. The analyser finds the chain $\text{char} \prec \text{int32} \prec \text{float}$, inserts the two intermediate cast instructions, and accepts the declaration. The runtime behaviour is: the byte value of ‘A’ (65) is zero-extended to the 32-bit integer 65, then lifted to the Q16.16 fixed-point representation $65 \times 65536 = 4259840$ by `F_I2F`. The declaration type-checks with zero diagnostics.

So transitivity holds constructively even though the relation is specified by only two base rules. The two-step chain is just the shortest derivation connecting `char` and `float` in the relation graph; that graph has three nodes and two directed edges, and the transitive closure has exactly three edges (the two base rules plus the derived `char` $\prec$ `float` step).

Exercise 5.4 (Symbol-table operations for the for loop). We walk through the analyser’s scope-stack operations from the moment it encounters the `for` keyword in the anchor program `math_fibonacci_iter` to the moment it finishes the loop body. Block 1 (the function body) is already open with five bindings: `n:int32`, `a:int32`, `b:int32`, `i:int32`, `t:int32`.

1. The analyser increments the loop-depth counter from 0 to 1 (not a scope-stack operation proper, but part of the `for`-node handler).
2. The init clause `i = 0` is processed. This is an expression, not a declaration, so no `insert` fires.
 - `lookup(i)` — found in block 1, type `int32`, lvalue. Returns the block-1 binding.
3. The guard clause `i < n` is processed.
 - `lookup(i)` — found in block 1, type `int32`.
 - `lookup(n)` — found in block 1, type `int32`.
4. The update clause `i++` is processed.
 - `lookup(i)` — found in block 1, type `int32`, lvalue (required for post-increment).
5. The analyser opens the loop body’s compound statement. **push** — block 2 is pushed as an empty frame on top of block 1.
6. The body `{ t = a + b; a = b; b = t; }` is processed. No declarations appear inside the braces, so no `insert` fires in block 2. The lookups, in source order:
 - `lookup(t)` — not in block 2, found in block 1.
 - `lookup(a)` — found in block 1.
 - `lookup(b)` — found in block 1.
 - `lookup(a)` — found in block 1.
 - `lookup(b)` — found in block 1.
 - `lookup(b)` — found in block 1.
 - `lookup(t)` — found in block 1.

7. The closing brace of the for-body causes the analyser to leave the compound statement.
pop — block 2 is popped.
8. The loop-depth counter is decremented back to 0.

Summary: one **push**, one **pop**, zero insert operations inside the loop body (the loop introduces no declarations in this program), and exactly nine lookup calls (one for *i* in the init, two in the guard, one in the update, and five in the body). Every lookup is satisfied in block 1; block 2 contributes nothing to name resolution here, but it is still pushed and popped because the compound-statement rule always maintains the stack.

Exercise 5.5 (Typing derivation for $(i < n) \parallel (a == 0)$). Assume $\Gamma = \{i : \text{int32}, n : \text{int32}, a : \text{int32}\}$. The derivation proceeds bottom-up from the leaves. FRISCCc uses the convention that integer literals type to `int32` and that a comparison of two `int32`-typed expressions produces `int32` — following C semantics, where a relational or equality operator yields the integer 0 or 1, not a distinct boolean type. The semantics layer has no `bool` type at all; the logical-or rule T-OR accepts two `int32`-typed operands and likewise returns `int32`. (The IR generator of Chapter 6 later lifts these integer results into a dedicated `bool` value when it lowers comparisons into conditional branches; at the semantics level they remain plain `int32`.)

Left sub-expression: $i < n$.

$$\frac{\frac{\Gamma(i) = \text{int32}}{\Gamma \vdash i : \text{int32}} \text{ T-VAR} \quad \frac{\frac{\Gamma(n) = \text{int32}}{\Gamma \vdash n : \text{int32}} \text{ T-VAR}}{\Gamma \vdash i < n : \text{int32}} \text{ T-CMP}$$

Right sub-expression: $a == 0$.

$$\frac{\frac{\Gamma(a) = \text{int32}}{\Gamma \vdash a : \text{int32}} \text{ T-VAR} \quad \frac{}{\Gamma \vdash 0 : \text{int32}} \text{ T-LIT}}{\Gamma \vdash a == 0 : \text{int32}} \text{ T-CMP}$$

Combining with logical-or.

$$\frac{\Gamma \vdash i < n : \text{int32} \quad \Gamma \vdash a == 0 : \text{int32}}{\Gamma \vdash (i < n) \parallel (a == 0) : \text{int32}} \text{ T-OR}$$

The derivation has six leaf nodes: two applications of T-VAR for *i* and *n*, one application of T-VAR for *a*, and one application of T-LIT for 0; plus two premises already appearing in the two comparison judgements. The two comparison subexpressions each require their operands to have matching `int32` types (satisfied here by the context), and produce `int32` results (0 or 1, per C semantics). T-OR then consumes two `int32`-typed values and produces `int32`. The whole expression type-checks with no implicit promotions: all four named values are already `int32`, and `int32` is the common numeric type for both comparisons.

The six-leaf count from the exercise statement matches: the derivation tree has four leaves (two T-VAR per comparison, one for *a*, one T-LIT) plus two interior T-CMP nodes plus one T-OR root. The figure [Figure 5.2](#) in the chapter uses the same schematic for the simpler *a + b* sub-expression; the same horizontal-bar notation with the rule name to the right applies here at every level.

F.6 Chapter 6: Lowering to typed IR

Exercise 6.1 (Lowering a nested expression by hand). We lower the expression $(x + 1) * (y - 2)$, where *x* is a local int variable at slot offset 0 and *y* is at offset 4. We apply the four-step pattern from [Section 6.4](#) at each operator node: (1) lower the left sub-expression to a handle, (2) lower the right sub-expression to a handle, (3) allocate a fresh temporary, (4) emit the instruction and return that temporary.

Lowering x: an identifier at a local slot is an addr/load pair. No arithmetic; the steps (3) and (4) for the `addr_of_symbol` and `load` opcodes fire internally:

```
t0 = addr_of_symbol local:x          ; step 1, addr
t1 = load t0 : int32                 ; step 1, load
```

Handle for *x* is *t1*.

Lowering the constant 1: a literal constant returns the inline handle `#1:int32` with no emitted instructions.

Combining at +: step (3) allocate *t2*; step (4) emit:

```
t2 = add t1, #1:int32 : int32        ; step 4, left-arm result
```

Handle for $x + 1$ is *t2*.

Lowering y: the same addr/load pair as for *x*:

```
t3 = addr_of_symbol local:y          ; step 2, addr
t4 = load t3 : int32                 ; step 2, load
```

Handle for *y* is *t4*.

Lowering the constant 2: inline handle `#2:int32`, no instructions emitted.

Combining at -: step (3) allocate *t5*; step (4) emit:

```
t5 = sub t4, #2:int32 : int32        ; step 4, right-arm result
```

Handle for $y - 2$ is $t5$.

*Combining at *: step (3) allocate $t6$; step (4) emit:*

```
t6 = mul t2, t5 : int32           ; step 4, top-level result
```

Handle returned to the caller is $t6$.

Complete emission sequence:

```
t0 = addr_of_symbol local:x
t1 = load t0 : int32
t2 = add t1, #1:int32 : int32
t3 = addr_of_symbol local:y
t4 = load t3 : int32
t5 = sub t4, #2:int32 : int32
t6 = mul t2, t5 : int32
```

Counts: seven temporaries total ($t0$ – $t6$). Four come from the two `addr_of_symbol/load` pairs (two for x , two for y). The remaining three ($t2$, $t5$, $t6$) hold arithmetic results.

Exercise 6.2 (Tracing a while-loop lowering). The fragment to lower:

```
int i = 0;
int s = 0;
while (i < 3) {
    s = s + i;
    i = i + 1;
}
return s;
```

The lowering walker allocates four blocks in the order the case handlers trigger.

L0 — entry block. Open when the function begins. The two declarations lower to `addr_of_symbol/store` pairs:

```
L0:
    t0 = addr_of_symbol local:i
    store t0, #0:int32 : int32
    t1 = addr_of_symbol local:s
    store t1, #0:int32 : int32
    jmp L1
```

The while handler fires: it allocates L1 (condition), L2 (body), L3 (exit), then seals L0 with an unconditional `jmp L1`.

L1 — condition block. Loads `i`, compares with the literal 3:

```
L1:
  t2 = addr_of_symbol local:i
  t3 = load t2 : int32
  t4 = cmp_lt t3, #3:int32 : bool
  br t4, L2, L3
```

Terminator: `br t4, L2, L3` — when `t4` is true (the condition holds) control goes to the body; otherwise it exits.

L2 — body block. The two assignments lower to load/add/store sequences:

```
L2:
  t5 = addr_of_symbol local:s
  t6 = load t5 : int32
  t7 = addr_of_symbol local:i
  t8 = load t7 : int32
  t9 = add t6, t8 : int32
  store t5, t9 : int32
  t10 = load t7 : int32
  t11 = add t10, #1:int32 : int32
  store t7, t11 : int32
  jmp L1
```

Terminator: `jmp L1` — this is the back-edge that reconnects the body to the condition.

L3 — exit block. Loads `s` and returns it:

```
L3:
  t12 = addr_of_symbol local:s
  t13 = load t12 : int32
  ret t13
```

Block and temporary counts: four blocks (L0–L3). Temporaries: `t0–t1` in L0 (two `addr/store` pairs), `t2–t4` in L1 (`addr/load/cmp`), `t5–t11` in L2 (two `addr/load/op/store` groups), `t12–t13` in L3 (`addr/load/ret`). Total: fourteen temporaries, of which six are `addr_of_symbol` results, five are `load` results, and three are computation results (one `cmp_lt` and two `adds`).

The back-edge is the `jmp L1` terminator of L2; it connects the body back to the condition. The loop-exit edge is the false branch of the `br` in L1: `br t4, L2, L3` routes to L3 when the condition fails.

Exercise 6.3 (Spot the well-formedness violation). (a) A block that contains `ret t2` followed by `store t0, #1:int32 : int32` violates **invariant 1**: a basic block must end in exactly one terminator as its final instruction, with no instructions after it. Here the `ret` terminator appears before the `store`, so the `ret` is not last and the `store` is in a position where no non-terminator instruction is legal. The invariant reads: “every block contains zero or more non-terminating instructions followed by exactly one terminator as the last line.” Placing a `store` after a `ret` breaks the “exactly one and last” contract.

(b) The instruction `t5 = add t3, #1:fixed16_16 : int32` violates **invariant 3** (operand-type consistency for arithmetic). The `add` opcode requires both operands to share the same arithmetic type. Here the first operand `t3` has type `int32` and the second operand `#1:fixed16_16` has type `fixed16_16`: a mismatch. The result-type annotation `int32` does not resolve this; operand types must agree before the result type is consulted. A legal form would be either `t5 = add t3, #1:int32 : int32` or `t5 = add t3, #1:fixed16_16 : fixed16_16`, depending on the intended precision.

(c) A function whose only block is `L0` ending with `jmp L0` must be checked against **invariant 5**: the entry block must have no incoming edges from elsewhere in the function. Here the single block `L0` is both the entry block and the target of the `jmp L0`: that `jmp` is an incoming edge to the entry block. The fragment therefore **does** violate invariant 5. The motivation: the back end emits the function prologue (frame allocation, register saves) once at the entry block; a back-edge into the entry would re-execute the prologue on each loop iteration, corrupting the frame.

(d) A function that declares `local w@8:int32` in its slot table but contains no `addr_of_symbol local:w` in its body violates the second half of **invariant 8** (slot-table coverage): every slot listed in the `.slots` section must be referenced by at least one `addr_of_symbol` instruction in the function body. The motivation given in [Section 6.5](#) is that a dead slot wastes frame space on the target machine; catching it early prevents the back end from allocating stack memory that the function never accesses. (The first half of invariant 8 is the converse: every `addr_of_symbol` must refer to a slot that exists in the table.)

Exercise 6.4 (Three-address verbosity audit). We work from [Listing 2](#), the pre-optimisation IR for `math_fibonacci_iter`. The listing appears in full in [Section 6.6](#); the real file is `book/listings/math_fibonacci_iter/intermediate_00.ir`.

The function `main` in that listing uses temporaries `t0` through `t21`: **22 temporaries** in total. Counting by defining opcode:

- `addr_of_symbol`: `t0` (n), `t1` (a), `t2` (b), `t3` (i) in `L0`; `t4` (i), `t6` (n) in `L1`; `t9` (a), `t11` (b), `t14` (t) in `L2`; `t17` (i) in `L3`; `t20` (a) in `L4`. That is **11** address temporaries.
- `load`: `t5` (load i), `t7` (load n) in `L1`; `t10` (load a), `t12` (load b), `t15` (reload b), `t16` (reload t) in `L2`; `t18` (load i) in `L3`; `t21` (load a) in `L4`. That is **8** load temporaries.
- `Arithmetic/comparison`: `t8` (`cmp_lt`), `t13` (`add`), `t19` (`add`). That is **3** computation temporaries.

Total: $11 + 8 + 3 = 22$. The 11 `addr_of_symbol` results and 8 load results together account for 19 of the 22 temporaries: only 3 hold actual arithmetic.

What if slot names were direct operands? If the IR allowed `t = add local:a, local:b : int32`, every `addr_of_symbol/load` pair for a read would collapse into a single named operand. Eleven address temporaries would disappear; the load temporaries that feed arithmetic would also disappear (their values would be inlined). The 3 arithmetic and comparison results would survive, leaving roughly **3–4** temporaries for this function.

Why the compiler chose the verbose form. As discussed in [Section 6.1](#), explicit load and store instructions make memory operations first-class citizens: the optimiser can reorder them, forward their results across multiple uses (`LoadForwardingPass`), and eliminate dead stores (`DeadSlotStoreEliminationPass`) using exactly the same uniform rewrite infrastructure it applies to any other instruction, with no special memory-operation casing. If slot names appeared directly as operands, the optimiser would need a separate analysis framework for “does this instruction read from or write to memory, and which slot?” — complexity that the explicit `addr_of_symbol/load/store` design simply does not have.

Exercise 6.5 (Tracing the slot table). We work with the slot table of the anchor’s `.func main():int32`, which contains five entries:

```
local n@0:int32
local a@4:int32
local b@8:int32
local i@12:int32
local t@16:int32
```

First: each slot is an `int32` (4 bytes), and the five slots are packed at offsets 0, 4, 8, 12, and 16. The highest offset is 16 and the slot is 4 bytes wide, so the total frame size is $16 + 4 = 20$ bytes. The directive that records this is `.frame locals=20 bytes align=4` at the top of the function header in [Listing 2](#).

Second: a sixth `int` local `u` declared after `t` would be assigned the next available offset, which is **20**, and the `.frame` directive would update to `.frame locals=24 bytes align=4`.

Third: if the optimiser eliminates all uses of `t` and removes it from the slot table, the remaining four slots retain their offsets (no renumbering is strictly required — the offsets are relative to the frame pointer and the frame itself shrinks). A compact updated `.slots` block with contiguous offsets would be:

```
.slots
  local n@0:int32
  local a@4:int32
  local b@8:int32
  local i@12:int32
```

If `t@16:int32` *remained* in the table after elimination, no instruction in the function body would contain `addr_of_symbol local:t`. That violates **invariant 8** (the second half: every slot listed in `.slots` must be referenced by at least one `addr_of_symbol` in the function body). The well-formedness checker in [Section 6.5](#) diagnoses this as a “slot declared but never addressed” error. The motivation is that a dead slot wastes four bytes of stack frame on the target machine — a space the back end allocates unconditionally because it trusts the slot table to be exact.

F.7 Chapter 7: Optimizing without breaking things

Exercise 7.1 (Trace constant folding by hand). We apply the passes in order, working through the fragment one step at a time.

After one pass of `TypedConstantFoldingPass`. The first instruction, `t0 = mul #3:int32, #4:int32 : int32`, has both operands as literal constants and `mul` in the foldable set, so the pass evaluates $3 \times 4 = 12$ and rewrites it to `t0 = #12:int32`. The second instruction, `t1 = add t0, #2:int32 : int32`, still has `t0` as a temporary — not a literal — so it is not foldable on this sweep. The remaining instructions are likewise untouched. The IR after one folding sweep is:

```

1      t0 = #12:int32
2      t1 = add t0, #2:int32 : int32
3      t2 = sub t1, #1:int32 : int32
4      t3 = cmp_lt t2, #20:int32 : bool
5      br t3, L_true, L_false

```

After one pass of `CopyPropagationPass`. The pass sees `t0 = #12:int32`, records the alias $t0 \mapsto \#12:\text{int32}$, and rewrites every subsequent use of `t0` in the block to the literal. The `add` instruction becomes `t1 = add #12:int32, #2:int32 : int32`. No further uses of `t0` exist, so the map is extended by $t1 \mapsto t1$ (not yet a literal).

```

1      t0 = #12:int32
2      t1 = add #12:int32, #2:int32 : int32
3      t2 = sub t1, #1:int32 : int32
4      t3 = cmp_lt t2, #20:int32 : bool
5      br t3, L_true, L_false

```

Counting pipeline iterations. After the first iteration (fold then propagate) the fragment still contains non-constant temporaries `t1`, `t2`, `t3`. *Iteration 2*: folding fires on `t1 = add #12:int32, #2:int32` producing `t1 = #14:int32`; propagation carries `#14` into the `sub`. Folding on the `sub` produces `t2 = #13:int32`; propagation carries `#13` into the `cmp_lt`. Folding on the `cmp_lt` evaluates $13 < 20 = \text{true}$ and produces `t3 = #true:bool`. *Iteration 3*: the fragment

now ends with `br #true:bool, L_true, L_false`, which folding cannot simplify further but which exposes a pattern for `ControlFlowSimplificationPass`: a branch on a constant Boolean is rewritten to an unconditional `jmp L_true`. After dead-temporary elimination removes the now-dead definitions, the whole fragment collapses to a single `jmp L_true`.

So the answer is **three pipeline iterations** before the constant-Boolean branch is visible, plus a final simplification step. The pass responsible for turning `br #true:bool, L_true, L_false` into `jmp L_true` is `ControlFlowSimplificationPass` (from [Section 7.3](#)), which handles the “branch on a constant Boolean” case by retargeting to the live successor and orphaning the dead one.

Exercise 7.2 (Which pass fired?). (a) Before: `t1 = mul t0, #1:int32 : int32`. After: `t1 = t0`.

Pass: `Int32ArithmeticPass` (from [Section 7.2](#)). The algebraic identity $x \times 1 = x$ holds in $\mathbb{Z}/2^{32}\mathbb{Z}$ regardless of the value of `t0`, so the instruction is replaced by a copy. Semantics are preserved because multiplying any integer by one changes no bits.

(b) Before: `store t0, #5:int32 : int32; t2 = load t0 : int32; t3 = add t2, #1:int32 : int32`. After: `store t0, #5:int32 : int32; t2 = #5:int32; t3 = add #5:int32, #1:int32 : int32`.

Pass: `LoadForwardingPass` (from [Section 7.4](#)). The pass tracks, for each slot, the most recently stored value. After the store of `#5:int32` to the address held in `t0`, the load of the same slot is redundant: the stored value is still current. The pass rewrites the load to `t2 = #5:int32`, a constant-bound copy. No intervening store touches the slot, so the forwarding is safe; the program observable behaviour is unchanged because the load would have produced `#5:int32` on any execution.

(c) Before: a basic block whose only instruction is `jmp L_step`, with two predecessor edges entering it. After: the block is deleted, and both predecessors now branch directly to `L_step`.

Pass: `ControlFlowSimplificationPass` (from [Section 7.3](#)), with the jump-only block case. A block containing nothing but `jmp L_target` contributes no work to the program; its only observable effect would be the program counter visiting it, which is not part of the IR’s observable behaviour. Every predecessor has its terminator retargeted directly to `L_step`, and the now-orphaned block is swept by `UnreachableBlockEliminationPass` on the next iteration.

(d) Before: `t10 = mul #8:int32, t9 : int32` inside a loop where `t9` is the induction variable, incremented by 1 per iteration. After: `t10` replaced by a fresh temporary initialised to 0 in the pre-header and incremented by 8 each iteration.

Pass: `InductionStrengthReductionPass` (from [Section 7.5](#)). The value `t9 * 8` changes by exactly 8 each iteration because `t9` changes by 1. The pass replaces the multiplication with an additive update, trading a call to `F_MUL` (or a shift) for a single `ADD` per iteration. Semantics are preserved because $t9_0 \times 8 + k \times 8 = (t9_0 + k) \times 8$ for any loop-count k , so

the running temporary tracks the same sequence of values that the original multiply would have produced.

Exercise 7.3 (Why does the pipeline terminate?). Define the measure $\mu(P)$ to be the pair

$$\mu(P) = (| \text{instructions in } P |, | \text{temporaries in } P |)$$

ordered lexicographically: $(a_1, b_1) < (a_2, b_2)$ iff $a_1 < a_2$, or $a_1 = a_2$ and $b_1 < b_2$. The pair is bounded below by $(0, 0)$ because neither count can be negative.

For the fifteen passes that only delete or rewrite, the argument is direct: each transformation either removes an instruction (decreasing the first component), removes a temporary (decreasing the second component without increasing the first), or changes neither — but in the last case the pass reports `changed = false` and the outer loop does not restart.

`InductionStrengthReductionPass` is the exception: it *inserts* instructions in the pre-header (an `ADD` and the initialisation of the fresh temporary) while removing a multiplication from the loop body. The inserted instructions are in the pre-header, which lies outside the loop and is executed at most once per invocation of the enclosing function; the removed instruction was inside the loop body and executed once per iteration. Although the raw instruction count may increase on the first application, the temporary count does not increase (the pass introduces one fresh temporary per strength-reduced multiply), and, more importantly, the `mul` instruction it replaces is not re-introduced on a later pass — the resulting loop body has one fewer multiplication, so the pass cannot fire on the same site again. The measure decreases strictly on each firing of each pass, and no pass can undo the reduction of another, so μ is a termination measure under the lexicographic order.

Exercise 7.4 (Reading the inner loop). The optimised inner-loop body from [Section 7.9](#) is:

```

1      t36 = addr_index t33, t35, 1    (1)
2      store t36, #0:int32 : char      (2)
3      t40 = add t37, t39 : int32      (3)
4      store t34, t40 : int32          (4)
```

(The address computations for `isPrime`, `j`, and `p` live in the pre-header after LICM, so they do not appear in the body's four instructions.)

Instruction (1) computes the element address `isPrime[j]`; it is the essential index arithmetic that identifies which composite to mark. *Instruction (2)* clears the cell; this is the loop's purpose. *Instruction (3)* adds `p` to the current value of `j`; this is the essential step advance that walks the sieve forward. *Instruction (4)* stores the updated `j` back so the next iteration reads the correct index. Every instruction is essential.

If a future pass specialised the inner loop for $p = 2$, the following further simplifications would become available. The `addr_index` element size is 1 (byte), so the index is not

scaled; no extra saving there. However, the step `add t40 = add t37, #2:int32` would have a constant second operand, which is already the case after constant propagation — the gain is nil if propagation has already fired. The real win is that the `addr_index` bounds check would become a comparison against a constant, allowing `AddrIndexAnalyzer` (or `ValueRangeSimplificationPass`) to prove the check redundant and eliminate it: with j starting at $p^2 = 4$ and the loop terminating when $j > 200$, the lower-bound check $j \geq 0$ holds vacuously and the upper-bound check $j < 201$ is enforced by the loop condition. Both guards could be removed, saving two comparisons and two conditional jumps per inner iteration. The eliminated guards would then make `UnreachableBlockEliminationPass` and `ControlFlowSimplificationPass` eligible to fire on the guard-exit edges.

Exercise 7.5 (The Hoare quote in context). The Knuth–Hoare warning targets *speculative* micro-optimisation: spending time on hot-path tuning before a profile says the path is hot, and thereby complicating code whose performance does not matter. Applied to the sixteen passes of this chapter, the warning does not bite, for two reasons.

First, the passes address redundancy that the lowering walker introduces *by construction*. The lowering walker of Chapter 6 is deliberately literal: it emits every memory read, every memory write, and every address computation that the source semantics requires, without looking sideways. The consequence is that the O0 IR of any program contains dead stores, redundant loads, loop-invariant address computations, and constant-foldable arithmetic — not because the programmer wrote bad code, but because structural literalness requires them. A pass that removes those is not guessing about hotness; it is correcting a known, systematic artefact of the compilation model.

Second, the passes are systematic, not speculative. Knuth’s original concern [49] was the programmer who says “I think this loop is slow” and manually unrolls it before running a profiler. The sixteen passes say: “every program that uses multiplication will pay `F_MUL`; every program that uses a loop will repeat address computations; these are not guesses but certainties derivable from the IR’s structure.” That is closer to the “critical 3 %” Knuth exempts from his warning than to the reckless 97 % he condemns.

In short: the sixteen passes perform (b), systematic removal of redundancy the lowering walker introduces by construction. They are not micro-optimisations on hot inner loops arrived at without profile data, and Knuth and Hoare would not object to them.

F.8 Chapter 8: Producing FRISC

Exercise 8.1 (Lower a binary operation by hand). We follow Algorithm 8.2. The IR instruction is `t7 = sub t2, t5 : int32`, with `t2` at `R5-10`, `t5` at `R5-14`, and `t7` at `R5-1C`.

Step 1: evaluate the *left* operand `t2` into `R0`. Since `t2` is a temporary (not a literal), `EVALVALUE` issues a load from its slot:

```
1 | LOAD R0, (R5-10) ; load t2 from its slot
```

Step 2: push it onto the stack to protect it while the right operand is computed:

```
1 | PUSH R0 ; save left operand
```

Step 3: evaluate the *right* operand t5 into R0:

```
1 | LOAD R0, (R5-14) ; load t5 from its slot
```

Step 4: move the right operand to R1:

```
1 | MOVE R0, R1 ; right operand -> R1
```

Step 5: recover the left operand from the stack:

```
1 | POP R0 ; restore left operand
```

Step 6: emit the FRISC subtraction (the sub opcode maps directly to SUB per Table 8.2):

```
1 | SUB R0, R1, R0 ; R0 = left - right
```

Step 7: store the result to t7's slot:

```
1 | STORE R0, (R5-1C) ; store result to t7
```

The complete sequence is seven instructions. Of those, **one** instruction performs the actual subtraction (SUB R0, R1, R0). The remaining **six** — two loads, one push, one move, one pop, one store — are the bookkeeping cost of having no register allocator: every temporary lives in a stack slot and must be shuttled through a register for each use.

Exercise 8.2 (Which fast path fired?). We follow the fast-path logic from Section 8.3.

Instruction	Fast path	Emitted FRISC
t1 = mul t0, #16	Power-of-two: $16 = 2^4$	SHL R0, 4, R0
t1 = mul t0, #1	Identity: $x \times 1 = x$	operand evaluated into R0
t1 = mul t0, #0	Zero: $x \times 0 = 0$	MOVE 0, R0
t1 = mul t0, #7	Small-constant open-coding ($7 = 111_2$)	shift-and-add sequence (see below)
t1 = mul t0, #11	None; falls back to F_MUL	CALL F_MUL
t1 = mul t0, t2	None (both operands variable)	CALL F_MUL

A note on case (b): the identity fast path emits no multiply at all. It simply evaluates the surviving (non-unit) operand straight into `R0` — a single `LOAD` or `MOVE` that the surrounding code would have performed anyway — and returns. There is no redundant `MOVE R0, R0` self-move, so the peephole pass of [Section 8.8](#) is not involved here; multiplication by 1 contributes nothing beyond materialising the operand.

Case (d), multiplying by 7, fires the small-constant open-coding path: 7 is one of the constants `FRISCcc` open-codes (`{3, 5, 6, 7, 9, 10, 12, 15}`), so instead of calling `F_MUL` it expands the multiply into a short shift-and-add sequence. Reading `7 = 1112` from the least significant bit, the generator accumulates $x + 2x + 4x$:

```

1      MOVE 0, R2          ; accumulator = 0
2      MOVE R0, R3         ; term = x
3      ADD  R2, R3, R2      ; acc += x      (bit 0)
4      SHL  R3, 1, R3       ; term = 2x
5      ADD  R2, R3, R2      ; acc += 2x     (bit 1)
6      SHL  R3, 1, R3       ; term = 4x
7      ADD  R2, R3, R2      ; acc += 4x     (bit 2)
8      MOVE R2, R0         ; result -> R0

```

A genuinely non-open-coded constant such as `#11` (not in that set) is what falls back to `CALL F_MUL`, as the final table row shows.

Exercise 8.3 (Materialise a comparison). Following [Algorithm 8.3](#), we lower `t9 = cmp_gt t8, #0 : bool`. Let the two generated labels be `L_CMP_TRUE_1` and `L_CMP_END_2`.

First, evaluate the left operand `t8` (say, from slot `R5-20`):

```

1      LOAD R0, (R5-20)     ; left operand t8
2      PUSH R0

```

Evaluate the right operand `#0` and move to `R1`; pop left:

```

1      MOVE 0, R0
2      MOVE R0, R1          ; right -> R1
3      POP  R0              ; left -> R0

```

Compare and jump to the true arm if the condition holds (`cmp_gt` maps to `JP_SGT`):

```

1      CMP  R0, R1
2      JP_SGT L_CMP_TRUE_1   ; R0 > R1 -> true
3      MOVE 0, R0           ; false arm
4      JP   L_CMP_END_2
5  L_CMP_TRUE_1
6      MOVE 1, R0           ; true arm

```

```

7  L_CMP_END_2
8  STORE R0, (R5-24) ; store t9

```

That is 10 instructions (excluding the slot load/store the caller provides), or 9 instructions for the materialisation itself.

With comparison-branch fusion. If the very next IR instruction is `br t9, L_pos, L_neg`, a fusing back end would collapse the materialisation and the branch into a single `CMP/JP` pair:

```

1  LOAD R0, (R5-20)
2  CMP R0, 0
3  JP_SGT L_pos
4  JP L_neg

```

That is 4 instructions instead of 10 for the pair: a saving of 6 instructions at this site. The materialised diamond (4 basic blocks) collapses to a straight-through `CMP` followed by two conditional jumps. FRISCcc does not fuse because it lowers each IR instruction in isolation, but the opportunity is real and the count makes it concrete.

Exercise 8.4 (Size a frame). We apply [Algorithm 8.1](#) to the given function. The inputs are:

- declared locals: $L = 12$ bytes,
- temporaries: `t0` through `t9`, so $tempCount = 10$,
- one call passing 2 arguments, so $maxArgs = 2$.

Locals area. $ALIGN4(12) = 12$ bytes.

Temporary area. $10 \times 4 = 40$ bytes.

Argument-scratch area. $2 \times 4 = 8$ bytes.

Raw total. $12 + 40 + 8 = 60$ bytes. Since 60 is already a multiple of 4, $ALIGN4(60) = 60$.

Zone	Size
Declared locals	12 bytes
Temporaries (<code>t0</code> – <code>t9</code>)	40 bytes
Argument scratch (2 words)	8 bytes
Total frame size	60 bytes

Slot of t_0 . Using the formula from [Algorithm 8.1](#):

$$\text{slot}[0] = \text{localsArea} + 4 \cdot 0 + 4 = 12 + 0 + 4 = 16.$$

Temporary t_0 lives at **16 bytes below the frame pointer**, written R5-10 in hexadecimal. Temporary t_1 is at offset 20 (R5-14), t_2 at 24 (R5-18), and so on.

Exercise 8.5 (When does the guard disappear?). We follow the logic of AddrIndexAnalyzer from [Section 8.5.1](#).

(a) **$a[0]$.** The index is the integer literal 0. The analyzer can prove at compile time: $0 \geq 0$ (lower bound holds) and $0 < 10$ (upper bound holds, given the array has 10 elements). **No guards are emitted.** The analyzer knows the index is in range without any runtime check.

(b) **$a[i]$ for a parameter i .** i is an unknown value arriving from the caller; the analyzer has no information about it. Both the lower-bound check ($i \geq 0$) and the upper-bound check ($i < 10$) are necessary and **both guards are emitted**: `CMP R1, 0; JP_SLT L_BOUNDS_ERROR` and `CMP R1, 0A; JP_SGE L_BOUNDS_ERROR` (0A is hex 10).

(c) **$a[k]$ where the preceding IR is $k = \#3$.** After the propagation passes have forwarded the constant #3 to every use of k , the analyzer sees a constant index of 3. As in case (a), $3 \geq 0$ and $3 < 10$ are both verifiable at compile time. **No guards are emitted.**

(d) **$a[n-1]$ for an unknown n .** The subexpression $n-1$ produces a temporary whose range the analyzer does not know (unless a prior ValueRangeSimplificationPass has bounded n). In the absence of range information, the analyzer is conservative: it cannot prove $n-1 \geq 0$ or $n-1 < 10$. **Both guards are emitted.** If a value-range analysis had established, say, $n \in [1, 10]$, then $n-1 \in [0, 9]$, both checks would pass, and the guards could be elided; but that requires the interval lattice to have been applied first, which is not guaranteed at the single instruction level.

F.9 Chapter 9: Runtime: helpers, traps, and frame layout

Exercise 9.1 (Trace a multiply). We hand-execute F_MUL from [Listing 4](#) on inputs $a = 6$, $b = 5$.

Sign handling. Both operands are non-negative ($6 \geq 0$, $5 \geq 0$), so neither negation fires and the sign flag R2 stays at 0. After the sign block: $R0 = 6$ (the shifting copy of a), $R1 = 5$ (the shrinking b), $R3 = 0$ (the accumulator).

Loop trace. The loop runs while $b \neq 0$ and tests bit 0 of R1 with `AND R1, 1, R4` each iteration. We enter the loop with $R3 = 0$.

Iter.	R0 ($a \ll k$)	R1 ($b \gg k$)	R3 (result)	ADD fires?
1	6	5	0	yes ($5 \& 1 = 1$): $R3 \leftarrow 0 + 6 = 6$
2	12	2	6	no ($2 \& 1 = 0$)
3	24	1	6	yes ($1 \& 1 = 1$): $R3 \leftarrow 6 + 24 = 30$
4	48	0	30	(exit: $b = 0$)

The loop runs **3 times** (b reaches 0 after three right-shifts). The ADD fires in **2 of those iterations** (iterations 1 and 3, corresponding to the set bits in $5 = 101_2$).

The sign flag R2 is still 0, so the negation guard does not fire. The final result is $R3 = 30$, which is moved to R6 and returned. $6 \times 5 = 30$ confirms the computation.

Exercise 9.2 (Reading hexadecimal immediates). The three immediates decode as follows.

Instruction	Value	Why correct
LOAD R1, (R5+0C) in F_MUL	$0C_{16} = 12$	The second argument is at R5+12: the saved R5 occupies (R5+0), the return address (R5+4), the first argument (R5+8), and the second argument at (R5+12) = 0C.
MOVE 20, R4 in F_DIV	$20_{16} = 32$	The restoring-division loop iterates exactly 32 times, once per bit of the 32-bit dividend; loading 32 into the counter register initialises the loop correctly.
SHL R0, 10, R0 in F_I2F	$10_{16} = 16$	Converting an integer to Q16.16 multiplies by 2^{16} , which is a left shift by 16; the immediate 16 (10 in hex) is the shift amount.

Exercise 9.3 (The cost of a guard). From Listing 5, a single bounds check consists of the following instructions (not counting the MOVE R0, R1 that copies the index into R1, since that belongs to the index computation itself):

```

1      CMP    R1, 0           ; lower-bound test
2      JP_SLT L_BOUNDS_ERROR  ; conditional jump
3      CMP    R1, 0C9         ; upper-bound test (201 decimal)
4      JP_SGE L_BOUNDS_ERROR  ; conditional jump

```

That is **4 instructions** added per array access (two CMP/JP pairs).

Loop cost. The initialisation loop writes `isPrime[i] = 1` for $i = 0$ to $i = 200$, which is 201 iterations. Each iteration performs one array access, so the bounds check fires 201 times. The extra instruction count is $201 \times 4 = 804$ instructions. For comparison, if the bounds-check elimination pass were able to recognise the loop pattern and prove $0 \leq i < 201$ at every access point, it would save all 804 of those instructions — a reduction of roughly half the extra overhead in the initialisation loop alone. The break-even point for the analysis pass is any loop that executes the checked access more than a handful of times; for a loop of even ~ 10 iterations, the saved 40 instructions outweigh any reasonable analysis cost.

Exercise 9.4 (Why zero division returns zero). **The case for returning zero.** FRISC has no exception mechanism and no operating system to deliver a signal to: the only “stop the world” primitive available is `HALT`. If `F_DIV` jumped to a trap like `L_BOUNDS_ERROR`, the program would halt unconditionally on any division-by-zero, regardless of whether the surrounding code intended to test and handle the case. Returning zero instead is a defined, if arbitrary, result that keeps the program running and allows the calling code to detect and respond to the error through ordinary conditional logic (`if (divisor == 0)`). It is also consistent with the “two’s-complement wrap-around” convention the rest of the arithmetic runtime adopts: when the exact mathematical result cannot be represented, the routine returns the most well-defined approximation it can, rather than halting. The choice matches what many hardware dividers do on division-by-zero when no trap is available.

The case against. Division by zero is almost always a programming error, and silently returning zero masks that error: a program that accidentally divides by zero will produce wrong answers and continue, potentially corrupting further state, rather than stopping at the point of the fault. A trap — even one as blunt as the `L_BOUNDS_ERROR HALT` — gives the debugger or test harness an immediate, unambiguous signal. For an array bounds violation the same argument applies, and FRISCcc does trap there, because the hardware’s behaviour (reading from a random memory address) is far more dangerous than zero. The inconsistency is a genuine tension in the design, and a production compiler would more likely trap in both cases and reserve silent returns for deliberate, documented saturating-arithmetic modes.

Exercise 9.5 (Fixed-point by hand). In the Q16.16 format, the stored integer for a value v is $\lfloor v \times 2^{16} \rfloor$ for positive values, using two’s-complement for negatives. We have $2^{16} = 65536$.

Stored representations.

Value	Stored integer	Hex
1.0	$1 \times 65536 = 65536$	0x00010000
0.25	$0.25 \times 65536 = 16384$	0x00004000
−2.5	$-2.5 \times 65536 = -163840$	0xFFD80000

For −2.5: the unsigned magnitude is $2.5 \times 65536 = 163840 = 0x00028000$; the two’s-complement negation is $2^{32} - 163840 = 4294803456 = 0xFFD80000$.

Verifying that Q16.16 addition is integer addition.

The stored integer for 0.25 is $16384 = 0x00004000$. Adding the two stored integers:

$$16384 + 16384 = 32768 = 0x00008000.$$

The stored integer for 0.5 is $0.5 \times 65536 = 32768 = 0x00008000$.

The two quantities agree: $0x00008000 = 0x00008000$. Addition of Q16.16 values is therefore identical to integer addition of their stored representations, because the scale factor 2^{16} distributes over addition:

$$(a \cdot 2^{16}) + (b \cdot 2^{16}) = (a + b) \cdot 2^{16}.$$

No correction step is needed, which is precisely why Q16.16 arithmetic is attractive on a machine like FRISC that has no floating-point unit: the ALU's integer ADD and SUB instructions work directly on the stored representations, requiring no extra hardware or software at all.

F.10 Chapter 10: Running it: the simulator

Exercise 10.1 (Follow the off-by-four). The unconditional $PC \leftarrow PC + 4$ in [Algorithm 10.1](#) runs *after* the handler. So the handler must write into PC a value that, after adding 4, equals the desired target. For a target of $0x1C0$ the handler must write $0x1C0 - 4 = 0x1BC$. In general, a CALL or unconditional JP to address A writes $A - 4$.

The bootstrap trace in [Table 10.1](#) confirms this. The CALL F_MAIN instruction sits at address $0x04$. The next instruction fetched is at $0x0C$ — the first instruction of F_MAIN. So the handler wrote $0x0C - 4 = 0x08$ into PC, and the +4 increment landed the program counter precisely on $0x0C$. The HALT at $0x08$ is skipped on the call and will be reached later, when main executes its RET.

Exercise 10.2 (Decode by hand). From [Algorithm 10.2](#), arithmetic and logic instructions carry the destination in bits 23–25, the first source in bits 20–22, the mode flag in bit 26, and the second source in bits 17–19 when the mode flag is 0. For CMP the situation is slightly different: CMP does not write a destination register (it writes only the flags), so the fields are used as two sources only. Reading the given word:

- Opcode field 01101 $\rightarrow$ CMP.
- First-source field 000 $\rightarrow$ R0.
- Mode bit 0 $\rightarrow$ register–register form.
- Second-source field 001 $\rightarrow$ R1.

The decoded instruction is `CMP R0, R1`. `CMP` computes $R0 - R1$ and discards the arithmetic result, keeping only the side effects on the status register. After the instruction, the flags are set from that subtraction:

- `Z` is set if $R0 = R1$ (the difference is zero).
- `N` is set if the difference's sign bit is 1 (i.e., $R0 < R1$ in signed arithmetic before overflow).
- `C` is set if there is an unsigned borrow out of bit 31 (i.e., $R0 < R1$ unsigned).
- `V` is set if the subtraction overflows in signed two's-complement.

A subsequent conditional jump like `JP_SLT` (signed less-than) reads the combination $N \neq V$ to decide the branch; `JP_EQ` reads only `Z`.

Exercise 10.3 (Predict the mix). `math_fibonacci_iter` computes the twentieth Fibonacci number via an integer loop over five local variables — no array, no struct, no aggregate, no global read. Its only memory traffic is loads and stores of those five integer slots. Compared with Figure 10.3 for the prime sieve:

The **memory category** (`LOAD`, `STORE`, `PUSH`, `POP`) should shrink most. The sieve's dominant cost was repeated reads of a 201-entry character array; Fibonacci has no such array. Its slot traffic is five `int32` locals accessed a fixed number of times per iteration, so the absolute `LOAD/STORE` count is small and grows only linearly with the loop bound, whereas the sieve's grows as $O(n^2)$ with the array bound.

Arithmetic (`ADD`, `SUB`) and control-flow (`JP`, `CMP`) instructions should occupy a larger *fraction* of the mix even if their absolute counts are modest, because memory traffic no longer crowds them out. The overall cycle count is much lower (the interpreter reports 478 IR steps for the anchor; the sieve needs 44,355 FRISC cycles), confirming that the computation is small.

Exercise 10.4 (Why does `LOAD` not move?). The sieve's `LOAD` instructions read each element of the `isPrime` character array, which is live data that changes during the run — a result of earlier `STOREs` marking composites. No local optimisation can prove those reads are redundant, because the values stored depend on runtime indices. Hence, load forwarding, dead-store elimination, and copy propagation find nothing to remove, and the `LOAD` count is identical at `-O0` and `-O1`.

A load *could* be eliminated if the loaded address is known at compile time to alias no store in the program. For example, a loop that reads a loop-invariant variable — one whose address is never passed to a write — could in principle have that load hoisted by the loop-invariant code motion pass, executing it once before the loop rather than once per iteration. FRISCcc does not currently perform this particular hoist on our anchor programs: even where a local is read but never rewritten inside a loop, its load survives every iteration in the `-O1`

output. The point stands as a description of what such an analysis *could* reach, not of a transformation the compiler applies here.

Exercise 10.5 (Where do the cycles go?). The sieve of Eratosthenes over an array of size N executes roughly $O(N \log \log N)$ inner-loop iterations, which is close to $O(N)$ for the sizes we care about. The benchmark array is $N = 201$ and uses 44,355 cycles. Scaling linearly, an array of size 10,000 would use on the order of $\frac{10,000}{201} \times 44,355 \approx 2,200,000$ cycles. An array of size 100,000 would reach roughly 22,000,000 cycles, still well short of the two-hundred-million watchdog. At $N \approx 900,000$ the linear estimate touches the cap — so to an order of magnitude, the harness can handle a sieve over an array up to $\sim 10^6$ entries before the watchdog fires.

The category that grows fastest as the array grows is **memory traffic** — specifically LOAD and STORE — because every inner-loop iteration reads and writes one element of `isPrime`. Each composite-marking pass over the array is one store per element; the counting pass is one load per element. The inner loop body is dominated by that memory access, so LOAD and STORE together scale with the total work, while control-flow instructions (JP, CMP) scale at the same rate but account for a smaller fraction.

F.11 Chapter 11: Interpreting instead of compiling

Exercise 11.1 (Counting steps by hand). We count executed steps — one per instruction and one per terminator — block by block, using the unoptimised IR from `book/listings/math_fibonacci_iter/intermediate_00.ir` and the execution counts derived from the loop bound $n = 20$.

Block	Runs	Steps/run	Total
L0	1	8 instrs + 1 term = 9	9
L1	21	5 instrs + 1 term = 6	126
L2	20	11 instrs + 1 term = 12	240
L3	20	4 instrs + 1 term = 5	100
L4	1	2 instrs + 1 term = 3	3
Total			478

Block L1 runs 21 times: 20 times taking the true edge into the body, and once taking the false edge to the exit block when `i` reaches `n = 20`. Blocks L2 and L3 each run exactly 20 times, once per loop iteration.

L2 contributes the most steps (240 out of 478, just over half), because it is the widest block and runs every iteration. It contains the three address-load pairs for `a`, `b`, and `t`; the add; the store to `t`; a second load of `b`; a store to `a`; a load of `t`; and a store to `b` — eleven instructions, one terminator, twenty times. The increment block L3 is narrower (five steps) and accounts for 100 of the remaining steps, while the condition block L1 accounts for 126.

Exercise 11.2 (Why optimisation saved exactly a hundred steps). Comparing `intermediate_00.ir` and `intermediate_01.ir`, the most important change is that the optimiser *merges block L3 into block L2*. In the unoptimised IR, after the body computes the next Fibonacci pair, control jumps unconditionally to L3, which loads `i`, adds 1, stores the result, and jumps back to L1. The optimiser recognises that this is an unconditional fall-through and folds L3’s increment into the end of L2. At the same time, load-forwarding removes the two redundant reloads (the former `t15` and `t16`) and the dead store of `t` (the former `t14`) from L2. The four increment instructions migrate in while four now-redundant instructions drop out, so L2 stays at exactly 11 instructions plus one terminator — the same width it had at O0 — while the L3 block no longer exists at all.

The saving is exactly $5 \text{ steps} \times 20 \text{ iterations} = 100 \text{ steps}$. The five steps are L3’s four instructions plus its `jmp L1` terminator. Each of those five steps was paid once per loop iteration; eliminating the block saves them all, twenty times over. This is the multiplicative leverage of loop-body optimisation: removing a single step from a block that executes k times saves k steps globally. The loop bound here is 20, so each instruction removed from the body saves exactly 20 steps — one instruction removed from L3 saves 20, and L3 had five, giving exactly 100.

Exercise 11.3 (Tracing a temporary’s two lives). Looking at block L1 in `intermediate_00.ir`:

```
t4 = addr_of_symbol local:i
t5 = load t4 : int32
```

`t4` is produced by `addr_of_symbol local:i`, which asks the interpreter to look up the slot named `i` in the frame’s `localAddresses` map and return the integer address at which that slot was allocated. The interpreter allocated slot `i` at offset $4 \times 3 = 12$ bytes from the base address `0x1000`, giving address `4108` (decimal). So `t4` holds an *address*.

`t5` is produced by `load t4 : int32`, which takes the integer `4108` stored in `t4`, uses it as a byte address into the interpreter’s `Memory` map, reads four little-endian bytes, and reassembles them into an integer. At the second entry into the loop condition, `i = 0`, so `t5 = 0`. That is a *value*.

Both sit in the same `Map<Integer, Integer>` (the frame’s `temps` map) as plain `Integers`. The interpreter never distinguishes them by type at run time; the distinction lives only in how the instruction that produced each one interpreted its operands.

When the interpreter runs `store t9, t15` in the loop body, the `Store` instruction carries an explicit *destination type* in its IR representation (here `int32`), and its first operand is always the address to write to. The IR defines `Store` as “write the second operand’s value into the memory address given by the first operand”. That structural contract on the instruction — which operand is the address and which is the value — is what settles the question, not a run-time type tag on the integers in the temporary map.

Exercise 11.4 (The bounds-check analysis). The analysis is a backward fixed-point over the data dependencies. It seeds the “flows-into-an-address” set with every temporary that appears as the *address operand* of a load or store.

In the given fragment, `t1` feeds a load, so `t1` is immediately in the set. `t1` was produced by `addr_index t0, ti, 4`, so the analysis marks that `addr_index` as checked.

`t3` feeds neither a load nor a store directly; it is only used in a pointer comparison. A pointer comparison reads the *value* of the temporary as an integer, not as a memory address that will be dereferenced. So `t3` never enters the set, and the `addr_index` that produces it is *not* checked.

Skipping the bounds check on `t3` is **safe**, not merely an optimisation. The only thing a bounds violation can do is cause the program to read from or write to memory outside the declared array. If the address is never dereferenced — only compared — then no invalid memory access occurs, and there is nothing for a bounds check to prevent. Inserting one would be redundant and would change the observable behaviour only for indices that are already out of range, for which the comparison’s result is well-defined regardless.

Exercise 11.5 (Where the two back ends may legitimately differ). The interpreter exhausts the **JVM thread stack**. Each call to an IR function becomes a recursive Java call to `interpretFunction`; each active call pushes a Java frame onto the JVM’s thread stack. The JVM enforces a per-thread stack size limit (typically 256 KB to 512 KB by default, configurable with `-Xss`), and when a deeply nested program exceeds it, a `StackOverflowError` is thrown from inside the interpreter, propagating up through all the recursive Java calls. The IR program never returns a value; it crashes the interpreter.

The compiled FRISC program exhausts the **FRISC stack in the simulator**. Each call executes the function prologue, which pushes the caller’s frame pointer and subtracts from `R7` to allocate locals. The stack grows downward from address `40000`. If `R7` underflows below the region where the global array `isPrime` (or any other data) lives, the next store corrupts data and the program produces a wrong answer or, if it overwrites the code itself, executes garbage instructions. The two limits are set by different rules: the JVM decides the interpreter’s limit; the memory layout decides the compiler’s.

A program that would trigger this difference is a function that recurses to depth D with a small local frame. If the JVM’s thread stack can hold at most $\sim 2,000$ Java frames for the interpreter but the FRISC memory can hold $\frac{40000}{\text{frame size}}$ FRISC frames — for a frame of 20 bytes that is 2,000 FRISC frames, roughly comparable — then the limits may coincide accidentally. But with a small frame and the default JVM stack the interpreter will fail at a shallower recursion than the FRISC compiled program. For instance, a trivial function with no locals (0-byte frame) recurses indefinitely on FRISC (until the stack wraps around), but the interpreter’s Java overhead per call is fixed regardless of frame size, so it fails at around 2,000–8,000 levels depending on the JVM.

This is **neither a bug in the interpreter nor a bug in the compiler**. The two back ends model different machines with different resources. The interpreter models a machine

with a Java-sized call stack; the compiled program models a FRISC machine with a 40 KB stack. Disagreement on *whether a program terminates* on a program that overflows the modelled resource is the expected behaviour of faithful modelling. The fix, if one is wanted, is to configure the JVM's thread stack with `-Xss` or to use the bytecode VM, which maintains its own explicit call stack and can be given an arbitrarily deep limit.

F.12 Chapter 12: A machine of our own

Exercise 12.1 (Counting the expansion). Block L1 in the prime-sieve bytecode (Listing 9) disassembles to the following eleven opcodes, at offsets 47 through 89:

Offset	Opcode
47	ADDR_SYM local:i
52	STORE_TEMP t2
57	LOAD_TEMP t2
62	LOAD_WORD
63	STORE_TEMP t3
68	LOAD_TEMP t3
73	PUSH_CONST #200
78	CMP_LE
79	STORE_TEMP t4
84	LOAD_TEMP t4
89	BR L2, L4

The corresponding IR block L1 contains three instructions and one terminator:

```

t2 = addr_of_symbol local:i      (1 IR instr)
t3 = load t2 : int32             (1 IR instr)
t4 = cmp_le t3, #200 : bool      (1 IR instr)
br t4, L2, L4                   (1 terminator)

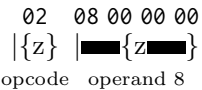
```

The expansion factor for this block is $11/4 = 2.75$, essentially matching the suite-wide average of ≈ 2.7 reported in Figure 12.4.

The single IR shape that expands into the most opcodes is **a load of a named slot** — specifically, the pattern `t = addr_of_symbol s` followed by `t' = load t : int32`, which becomes five stack opcodes: `ADDR_SYM`, `STORE_TEMP`, `LOAD_TEMP`, `LOAD_WORD`, `STORE_TEMP`. The reason is that the stack machine must explicitly name every temporary on the way in and on the way out; the three-address IR does both in one instruction by naming the destination. A comparison like `cmp_le t3, #200` also expands to four opcodes (`LOAD_TEMP`, `PUSH_CONST`, `CMP_LE`, `STORE_TEMP`), so the slot-load pattern is the wider of the two.

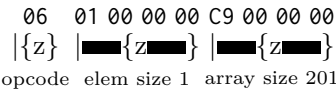
Exercise 12.2 (Reading the byte encoding). Using the one-opcode-byte plus four-little-endian-operand-bytes scheme of [Figure 12.1](#):

STORE_TEMP 8 (opcode byte 02): the operand is the integer 8, which in 32-bit little-endian is 08 00 00 00. The full encoding is:



Five bytes total.

ADDR_INDEX_CHK with element size 1 and array size 201 (opcode byte 06): the two operands are 1 and 201. In little-endian, 1 is 01 00 00 00 and 201 (which is 0xC9) is C9 00 00 00. The full encoding is:



Nine bytes total (one opcode byte plus two four-byte operands).

Little-endian order puts the least-significant byte at the lowest address, matching the byte-addressable memory convention the VM uses for all word storage (storeWord and loadWord both treat the byte at the lowest address as the low eight bits of the word). Using the same convention for operand bytes means the program counter can advance through the bytecode stream with a single readInt helper that assembles a word from the next four bytes, with no special-casing for little-endian versus big-endian.

Exercise 12.3 (Why two stacks). From the VM trace in [Listing 11](#), steps 17 through 22 are:

Step	Opcode	Effect	Stack depth
17	LOAD_TEMP 3	push value of t3 (which is 0)	0 → 1
18	PUSH_CONST 200	push the constant 200	1 → 2
19	CMP_LE	pop two, push result (1)	2 → 1
20	STORE_TEMP 4	pop result into t4	1 → 0
21	LOAD_TEMP 4	push value of t4	0 → 1
22	BR 622 751	pop condition, branch	1 → 0

The stack ends at depth 0, exactly where it began at step 17. The two pushes at steps 17–18 are consumed by the comparison at step 19; the result is named into a temporary at step 20 so the stack is clean; then the temporary is reloaded and consumed by the branch. The operand stack is a scratchpad that rises and falls within the expression and returns to baseline at the boundary of each IR statement.

If the VM used a **single stack for both operands and call frames**, a CALL instruction would have to push its new activation record onto the same array that currently holds mid-expression operand values. When the callee eventually executes a RET and pops its

frame, the operand values that were below the frame would need to be accessible to the caller — but the RET would need to know exactly how many frame entries to pop without touching them. Since the number of live operands on the stack at the call site varies instruction by instruction, there is no static frame size the VM could rely on. Any design that interleaves frame records and operand values in a single stack requires the callee to carry a “base pointer” that records where its frame starts within the operand region — at which point the design has effectively reinvented a separate frame stack, just encoded inside the same array. The JVM, in fact, gives each *frame* its own operand stack precisely to avoid this entanglement.

Exercise 12.4 (Where the back ends may differ). The **interpreter** exhausts the **JVM thread stack**: each IR call becomes a recursive Java call to `interpretFunction`, which pushes a Java frame. The JVM’s per-thread stack depth limit (set by `-Xss`, defaulting to 256–512 KB) determines how many recursive calls can be in flight simultaneously. For a typical JVM and a typical IR call with a non-trivial frame, the limit is in the range of a few thousand recursive calls.

The **VM** exhausts **its own call stack**: the explicit `ArrayDeque` (or equivalent) that CALL pushes frames onto and RET pops. The VM sets this limit independently of the JVM’s thread stack; by default FRISCCc’s VM stops at sixteen million dispatched instructions (a watchdog on total work, not on recursion depth), but a deliberate call-stack limit can be added and can be made larger or smaller than what the JVM allows.

To measure each limit empirically, write a program:

```
int recurse(int n) { return n == 0 ? 0 : recurse(n - 1) + 1; }
int main() { return recurse(DEPTH); }
```

Increase DEPTH until the back end fails. The interpreter will throw a `StackOverflowError` from Java; the VM will either trip its watchdog or throw its own “call stack overflow” diagnostic.

On a typical JVM with default settings, the interpreter exhausts its thread stack at around 2,000–8,000 levels of IR recursion (the overhead per Java frame for `interpretFunction` is substantial — it includes the `Frame` object, the temporary map, and the block iterator, all on the Java heap but with each recursive call adding a Java frame of its own). The VM, owning its call stack, can be configured to sustain much deeper recursion; it would typically survive a depth of 10,000 or more before any limit is hit. So **the VM survives a deeper recursion** on a typical JVM without tuning.

Exercise 12.5 (The cost of a checked access). If a program indexes the same array a thousand times in a hot loop, the VM executes the bounds comparison inside `ADDR_INDEX_CHK` **a thousand times** — once per dispatch, at run time. The comparison is cheap (two integer compares and a trap), but it happens as often as the loop iterates.

The analysis that *decided* to emit `ADDR_INDEX_CHK` rather than plain `ADDR_INDEX` ran **exactly once**, at lowering time, before the program began executing. The lowerer walked the function’s IR, computed the fixed-point set of temporaries that flow into memory addresses, identified this `addr_index` as one whose result is dereferenced, and encoded that decision as a single opcode choice.

By contrast, the **interpreter** re-runs the same fixed-point analysis **every time it enters the function** — once per call, not once per array access. For a function called once, the interpreter performs the analysis once and then checks the index a thousand times, the same as the VM. For a function called C times, the interpreter runs the analysis C times while the VM always runs it once (at lowering) and performs $C \times 1,000$ run-time checks.

What this says about where the two designs spend their effort: the interpreter distributes its one-time analytical cost across every call, whereas the VM pays that cost exactly once and stores the result in the bytecode permanently. The VM’s inner loop is therefore narrower — it contains no structural analysis, only opcode dispatch — which is precisely the “compute the facts that do not change once, and write them down” principle of [Section 12.3](#).

F.13 Chapter 13: Case studies

Exercise 13.1 (Breadth versus depth). We verify $C(20) = 21,891$ by the recurrence $C(n) = 1 + C(n-1) + C(n-2)$ with base cases $C(0) = C(1) = 1$. The first few values are

$$C(2) = 3, \quad C(3) = 5, \quad C(4) = 9, \quad C(5) = 15, \quad C(6) = 25, \quad C(7) = 41, \quad \dots$$

Continuing the recurrence to $n = 20$ yields $C(20) = 21,891$, which matches the chapter’s step-counted figure exactly. The sequence grows because each term is the sum of its two predecessors plus one, so it tracks the Fibonacci numbers themselves: $C(n) = 2F(n+1) - 1$, where F is the ordinary Fibonacci sequence. (This identity can be proved by a straightforward induction and accounts for the additive “+1” at each step.)

The *depth* of the recursion for `fib(20)` is 20: the longest root-to-leaf path in the call tree passes through the left branch at every level, reaching `fib(19)`, `fib(18)`, ..., `fib(0)`, a chain of twenty-one frames including the root call itself from `main`. The maximum call stack depth is therefore 20 live `fib` activations (plus `main`), exactly as [Figure 13.1](#) depicts.

Now suppose we doubled the depth by computing `fib(40)` instead. The breadth grows with the Fibonacci numbers, which increase roughly by the golden ratio $\phi \approx 1.618$ per step. Doubling the depth — going from $n = 20$ to $n = 40$ — multiplies the breadth by approximately $\phi^{20} \approx 15,127$, not by 2. In absolute terms, $C(40) = 2F(41) - 1 = 331,160,281$. The stack height grows by 20 extra frames (a modest linear increase in space), while the call count swells by a factor of roughly fifteen thousand (an exponential increase in time). Breadth and depth are linked by the recurrence but grow at completely different rates: depth is the recursion’s spatial cost, breadth is its temporal cost, and it is the exponential growth of breadth that makes naive recursive Fibonacci impractical for large n .

Exercise 13.2 (Reading the helper budget). The chapter reports that the three arithmetic helpers together consume 76% of GCD’s 1,457 executed FRISC instructions, with per-routine shares of 51.9% for `F_MOD`, 19.9% for `F_DIV`, and 4.3% for `F_MUL`. Converting to absolute instruction counts:

- `F_MOD`: $0.519 \times 1,457 \approx 756$ instructions.
- `F_DIV`: $0.199 \times 1,457 \approx 290$ instructions.
- `F_MUL`: $0.043 \times 1,457 \approx 63$ instructions.

The remaining 24% — roughly 350 instructions — covers the actual `F_GCD` logic and `main`.

To estimate how many times `F_MOD` was called, we need an approximate per-call instruction count. Listing 17 shows the shift-and-subtract core: the loop body runs 32 iterations (`MOVE 20, R4` loads the bit counter in hexadecimal, giving decimal 32), with roughly four to six FRISC instructions per iteration, plus sign-handling preamble and epilogue. A reasonable estimate is 80–120 FRISC instructions per call, say ≈ 100 on average. Dividing: $756/100 \approx 7\text{--}8$ calls to `F_MOD`.

Now we check against Euclid. The program computes `gcd(84, 30)` and then `lcm(84, 30) = (84 / gcd) * 30`. Tracing Euclid on the inputs 84 and 30:

$$84 \bmod 30 = 24, \quad 30 \bmod 24 = 6, \quad 24 \bmod 6 = 0.$$

Three iterations, three calls to `F_MOD`. Then `lcm` performs one `/` (one call to `F_DIV`) and one `*` (one call to `F_MUL`). So the program makes **three** `F_MOD` calls, **one** `F_DIV` call, and **one** `F_MUL` call.

The estimate of 7–8 calls overshoots, suggesting the per-call cost of `F_MOD` is higher than 100 instructions — around $756/3 = 252$ per call, which is consistent with a sign-handling prologue plus the 32-iteration loop at roughly six to eight instructions per step plus epilogue. The estimation exercise reveals that the instruction cost per helper call is the quantity to nail down; once you have it, every other cell of the helper budget follows.

Exercise 13.3 (Predicting a cell). We use the two factors the chapter reports for `-00` knapsack: a VM expansion of ≈ 2.69 opcodes per interpreter step, and a native expansion of 4.25 FRISC instructions per step.

VM dispatch prediction.

$$1,563 \times 2.69 = 4,204.5 \approx 4,205.$$

The table records exactly 4,205. The prediction is off by less than one opcode.

Native instruction prediction.

$$1,563 \times 4.25 = 6,642.75 \approx 6,643.$$

The table records exactly 6,643. Both predictions match the measured values to within rounding.

The VM prediction is more accurate in general because the VM expansion factor is a *structural constant* of the IR-to-stack lowering: each three-address IR instruction compiles to a fixed pattern of `LOAD_TEMP`/operand-push and result opcodes, regardless of what the instruction computes. A multiply and an add lower to the same number of stack opcodes because the lowerer emits one opcode per operation and one push per operand, and the operand count is fixed for three-address form. The native expansion factor, by contrast, is *program-dependent*: it reflects how many of the IR operations are software-emulated helpers (each expanding to tens or hundreds of FRISC instructions) and how many are hardware-native (each expanding to a handful). Knapsack has zero helpers, which is why its native factor is the lowest in the chapter. A program with a high helper share — like GCD’s $20.2\times$ — cannot be predicted from the step count alone; you also need to know the target’s instruction set.

Exercise 13.4 (When folding wins). The precise property that allows constant folding to fully collapse kinematics is that *every operand of every operation is a compile-time constant*. The source defines x_0 , v_0 , a , and t as literal values, so the lowering walker emits store-then-immediately-load sequences for constants that are known at compile time. The typed constant folding pass (Chapter 7) propagates those constant values forward through the expression dag: $v_0 t$ is a constant because both are constants, so $x_0 + v_0 t$ is a constant, and so on up the entire formula. The optimiser evaluates the whole computation once, at compile time, in Q16.16 arithmetic, and replaces twenty-three instructions with a single `ret #5.5:float`. The key property is therefore: **all operands are literals, so the entire computation is pure constant-propagation fuel**.

Recursive Fibonacci has no constants of consequence: the argument $n = 20$ is passed in, not hardcoded, and the recurrence `fib(n-1) + fib(n-2)` has operands that depend on prior recursive calls whose results are not known until run time. There is no constant sub-expression the optimiser can fold; the program’s redundancy is algorithmic — it recomputes intermediate values — but that redundancy is invisible to a local, semantics-preserving rewrite. The key property is therefore: **all operands are runtime-varying, so constant folding finds nothing to do**.

Now consider a variant of kinematics in which exactly one operand is a runtime input, for example:

```
int main() {
    float x0 = 0.5;
    float v0 = 1.5;
    float a = 1.0;
    float t;           /* read from the caller */
    t = input();        /* hypothetical runtime input */
    return x0 + v0 * t + 0.5 * a * t * t;
}
```

Because t is unknown at compile time, neither $v0 * t$ nor $a * t * t$ can be folded. The sub-expressions that involve only the four literal constants ($x0$, $v0$, a , 0.5) might reduce slightly, but the multiplies that involve t survive. Of the original twenty-three IR instructions, only a few constant-only sub-trees fold; the majority remain. The -01 interpreter step count will be **closer to 23** than to 1: perhaps in the range of 15–20 steps, depending on how many constant sub-expressions share no dependency on t , but nowhere near the single-step collapse that all-literal kinematics achieves.

Exercise 13.5 (A fairer speed question). The chapter’s counts — executed FRISC instructions, interpreter steps, VM dispatches — measure *work*: primitive operations performed by each machine to run the program. A wall-clock comparison is a different and harder question, and making it defensible requires holding several things constant.

First, you would need to hold the *host hardware* constant. The native pipeline runs on `friscjs`, a JavaScript FRISC simulator that itself executes on a Node.js runtime on a physical CPU. The interpreter and VM run on the JVM. Comparing milliseconds across these two runtimes conflates the cost of the program’s computation with the cost of the host’s just-in-time compiler, garbage collector, scheduler, and CPU cache state. Timing the JavaScript simulator against the JVM interpreter would be misleading because any difference in the numbers would reflect differences between the Node and JVM runtimes at least as much as differences between the three back ends.

Second, you would need to hold the *measurement conditions* constant. Mytkowicz and colleagues [63] showed that wall-clock measurements of binaries can vary by more than the speed-up being measured depending on factors as innocent as link order and environment-variable size. On a setup with a JavaScript simulator, the JIT warmup time for the simulator’s own hot dispatch loop would further pollute any timing.

The measurement that would *fairly* capture the native pipeline’s intended advantage is not a wall-clock comparison on the current stack but a count of executed FRISC *cycles* on a cycle-accurate FRISC hardware model (or, more practically, on silicon). On real hardware, the native code needs no interpreter overhead per step, no JVM, and no JavaScript runtime: it runs at the full clock frequency. Counting cycles rather than instructions, against a hardware specification rather than a simulator implementation, would let you compare all three back ends on the same axis.

The distinction the chapter draws is this: work counts measure the *shape* of each back end’s cost — where the operations go and how the program’s structure determines that shape — which is a durable, host-independent property. A millisecond reading, by contrast, measures the shape *convolved* with the performance characteristics of the entire software stack below it, and on a stack as tall as ours that convolution obscures far more than it reveals. Counts let us study the compiler; a clock would make us study the environment.

F.14 Chapter 14: Performance

Exercise 14.1 (The weighted-mean trap). From Table 14.2, recursive Fibonacci executes 1,368,194 instructions at both -00 and -01 — its dynamic saving is exactly zero. The full suite totals are:

suite -00 = 1,547,690 executed instructions,
 suite -01 = 1,544,567 executed instructions,
 saving = 3,123 instructions = 0.20%.

Removing recursive Fibonacci from both totals:

remaining -00 = 1,547,690 − 1,368,194 = 179,496,
 remaining -01 = 1,544,567 − 1,368,194 = 176,373,
 saving = 179,496 − 176,373 = 3,123 instructions,
 saving % = $\frac{3,123}{179,496} \approx 1.74\%$.

The saving in absolute instructions is *identical* in both calculations — 3,123 — because Fibonacci contributes equally to the numerator and denominator when it is included. What changes is the denominator: including a million-plus-instruction program that the optimiser cannot touch at all dilutes the percentage from 1.74% to 0.20%.

The two numbers differ so drastically because the suite-wide aggregate is an *execution-weighted arithmetic mean*, and a single unusually large program dominates it. The 0.20% figure is honest about the optimiser’s aggregate impact on this particular corpus — but it answers a different question than “how much does -01 help on programs where it can do something?” The 1.74% answers that question.

Neither number is wrong; they measure different things. To someone deciding whether to enable -01 in practice, the more informative pair is the *per-program* column of Table 14.2, which shows 1.0×–1.18× across thirteen programs and 21.6× for the one program (kinematics) whose structure aligns perfectly with the constant folder. The aggregate hides exactly the variance that determines whether -01 helps your program, which is the trap the exercise is named for.

Exercise 14.2 (Static versus dynamic). GCD/LCM at -01 reduces its IR by 17% (from 41 to 34 lines) while its dynamic instruction count falls by only 3.7% (from 1,457 to 1,404), giving a ratio near 1.04.

The explanation lies in *where* the removed IR lines execute. The optimiser is good at eliminating redundancies that the lowering walker emits in straight-line setup code: a slot address loaded and then immediately reloaded, a copy introduced to hold an argument, a store whose value is never read. These instructions appear in the function’s entry block and

in the return sequence, and they each run exactly *once* per call. The GCD program makes one call to `gcd` and one to a short `lcm` wrapper — so the setup and teardown sequences run a handful of times in total.

The hot work of the program is the Euclid loop, which performs three `F_MOD` calls, one `F_DIV`, and one `F_MUL`. Every one of those calls is an executed instruction the optimiser cannot remove, because the `mod` operator is genuinely needed — its result is used — and there is no constant to fold, no copy to eliminate, and no invariant to hoist. The five helper calls alone account for the overwhelming majority of the 1,404 executed instructions.

So the optimiser deletes several IR lines, each of which runs once (saving ~ 53 executed instructions, the difference between 1,457 and 1,404), while the instructions that run the most — the helper bodies invoked repeatedly inside `F_MOD`'s 32-iteration loop — are untouched. A pass can remove a sixth of the IR because the IR has many lines that execute only once; it removes only a twenty-fifth of the work because the work is concentrated in a few lines that execute many times. This is the static-versus-dynamic split stated precisely: instruction count in the text tells you *how many* instructions the compiler emits; executed-instruction count tells you *how often* those instructions run, and only the second number reflects speed.

Exercise 14.3 (Reading the ablation). The eight passes with no run-time effect on the suite are: `CastSimplificationPass`, `Int32ArithmeticPass`, `Int32ShiftPass`, `ValueRangeSimplificationPass`, `LoopInvariantCodeMotionPass`, `InductionStrengthReductionPass`, `TinyFunctionInliningPass`, and `UnreachableBlockEliminationPass`.

We take `LoopInvariantCodeMotionPass` (LICM) as our example.

LICM hoists expressions whose operands do not change inside a loop out of the loop body and into the loop preheader. On our suite it fires on several programs with loops — Chapter 7 shows it removing lines of IR from the prime sieve and the knapsack solver, among others — yet removing it from `-O1` changes the suite's executed-instruction count by exactly zero (Table 14.3).

The reason is that what LICM hoists on these particular loops is already cheap in a way the instruction count captures but the hoisting does not improve. The invariant expressions LICM moves are typically address computations or simple copies: on the FRISC, an address formed from a base pointer and a constant offset costs one instruction whether it is inside the loop or hoisted outside it, because the back end can fold constant offsets into addressing modes at no extra instruction. Moving the computation outside the loop shrinks the IR (fewer lines in the loop body) but does not change how many FRISC instructions the loop executes on each iteration.

LICM nonetheless earns its place in the pipeline on two other grounds. First, it is a *size* win: code that executes once in the preheader instead of N times in the body is literally smaller, which benefits instruction-cache pressure in larger programs even if our suite is too small to show it. Second, and more importantly for compiler correctness, hoisted expressions expose further opportunities for downstream passes: a value moved to the preheader is more likely

to be recognised as dead inside the loop by `DeadTempEliminationPass`, and it is a better candidate for `CopyPropagationPass` because it is now defined exactly once in a dominator of all uses. A pass that earns nothing on the dynamic metric may still be indispensable as a *normalisation step* that puts the IR in the canonical form the next pass expects.

Exercise 14.4 (Helper arithmetic). From Table 14.2, CRC checksum executes 89,828 instructions at -O1. The chapter reports 85.7% of those in `F_MOD`:

$$\begin{aligned}\text{helper instructions} &= 0.857 \times 89,828 \approx 76,982, \\ \text{user instructions} &= 89,828 - 76,982 = 12,846.\end{aligned}$$

We need the number of `F_MOD` calls. The chapter describes `F_MOD` as a 32-iteration shift-and-subtract loop with a few instructions per iteration plus sign-handling and call overhead; we estimated approximately 150–260 FRISC instructions per call in Section F.13. Using the GCD calibration from that exercise — roughly 252 instructions per `F_MOD` call — we get an estimated call count of $76,982/252 \approx 305$ calls to `F_MOD`. (The exact count depends on the CRC input data; this is an order-of-magnitude estimate.)

Now suppose a hardware modulo instruction costing 4 cycles replaces the entire `F_MOD` helper. Each of the ≈ 305 calls is replaced by one 4-cycle instruction. The new helper “cost” is $305 \times 4 = 1,220$ cycles, replacing the current 76,982. The new total executed-instruction count (under the assumption that the user code and call overhead are unchanged) is:

$$12,846 + 1,220 \approx 14,066 \text{ cycles.}$$

The speed-up relative to the original 89,828:

$$\text{speed-up} \approx \frac{89,828}{14,066} \approx 6.4 \times .$$

Now compare to the *Amdahl ceiling* [5]. With $f = 0.857$ of the program in the helper, the maximum possible speed-up from making the helper infinitely fast is:

$$\frac{1}{1-f} = \frac{1}{1-0.857} = \frac{1}{0.143} \approx 7.0 \times .$$

Our estimated $6.4\times$ is close to but below that ceiling, because we replaced the helper with a cheap 4-cycle instruction rather than making it free. The key takeaway is that even a perfect hardware fix — one that drove `F_MOD` to zero cost — could at most improve CRC checksum by $7.0\times$, no matter how clever the compiler; and our hardware revision comes within 91% of that ceiling ($6.4/7.0$). Amdahl’s law makes the ceiling visible even before you know the exact call count: $f = 0.857$ tells you that 85.7% of this program’s work is in the helper, and nothing done to the other 14.3% can move the overall speed-up past $7\times$.

Exercise 14.5 (Convergence by hand). Consider a program that requires two productive sweeps. After the first sweep, some passes have changed the IR. But those changes may have *created new opportunities* for other passes that already ran earlier in the same sweep: dead-temp elimination can only remove a copy once copy propagation has substituted the copy’s right-hand side into all uses, and if copy propagation ran after dead-temp elimination in the same sweep, the newly dead copy will not be removed until the next sweep. In general, a change made by pass i in sweep k may enable pass $j < i$ to do additional work — work that pass j could not do when it ran earlier in the *same* sweep because the enabling change had not happened yet.

To predict ahead of time how many sweeps are needed, you would have to know, for every pass j in the list, whether any pass $i > j$ (or the combination of multiple passes) in the current sweep will produce an IR change that enables j to change something in the next sweep. This is equivalent to computing the transitive closure of the pass-interaction graph under a particular input program — a problem that is, in general, as hard as running the optimiser. There is no efficient static analysis that can read the program text and say “two productive sweeps will suffice”; the pipeline must discover that answer empirically by re-running the pass list and checking whether each sweep was productive.

The property that guarantees the loop stops at all is the existence of a *well-ordered termination measure* on the IR, one that each pass either strictly decreases or leaves unchanged. Every pass in the -O1 pipeline either reduces the number of IR instructions (by eliminating dead code, folding constants, or simplifying control flow) or leaves the instruction count the same and reduces some secondary measure (such as the number of distinct temporaries, or the nesting depth of a branch structure). Because these measures are bounded below by zero, the pass list cannot keep finding productive changes forever: a sequence of sweeps each of which reduces the measure must terminate in finite steps. The five-sweep cap is a *safety valve* that bounds the compile-time cost in pathological cases — like the kinematics formula that cascades constant folding through a deep expression chain — without relying on the termination argument alone. As the chapter notes, at the cap the IR is still correct even if it is not yet at a fixpoint; the cap trades a guaranteed fixpoint for a guaranteed bound on compile time.

Appendix G

Formal Proofs

This appendix collects the formal proofs that support the chapter narrative. Semantic-preservation proofs for FRISCCc’s sixteen optimization passes occupy the bulk of it, but parser correctness and other metatheoretic results live here as well: anywhere a chapter sidebar states a theorem and defers the proof, this is where the deferred proof lands. The shared machinery comes first — what “semantic preservation” means, the small-step semantics of the IR, the observation function, and the simulation lemma that every per-pass proof reuses — so that each pass proof need only exhibit its own simulation relation and discharge the local cases.

G.1 What “semantic preservation” means here

Every optimization pass in [Chapter 7](#) rewrites one IR program into another, and every such rewrite is asked to do exactly one thing: leave the program’s observable behaviour alone while it changes the program’s text. The proofs that follow each discharge that obligation for one pass. Rather than re-establish the meaning of *observable behaviour* in sixteen separate places, we fix it here, once, together with the operational semantics it is defined against and the simulation machinery the per-pass proofs reuse. The treatment is standard — it follows the structural-operational style of Plotkin [70] and Winskel [84], and the compiler-correctness framing of Leroy’s CompCert [56] and Appel’s textbook [7] — specialized to the small, fully typed, I/O-free IR that FRISCCc actually manipulates.

The object language

The passes operate on FRISCCc’s typed three-address IR, whose full abstract syntax is given in [Appendix D](#). We summarize only the structure the proofs depend on.

A *program* P is a collection of named functions over an optional set of typed globals and named struct types. A *function* f carries a return type, a list of typed parameters, a frame declaration fixing the size and alignment of its local-storage region, a *slot table* mapping each parameter, local, and spill name to a type and a byte offset within that frame, and a non-empty list of *basic blocks*. Each block is a labelled, straight-line sequence of *instructions* ending in exactly one *terminator*. The first block of f is its entry.

Instructions come in three non-terminating shapes and three terminating ones. The non-terminating instructions are the temporary assignment $tn = r$ (binding temporary tn to the value of right-hand side r), the typed store $\text{store } a, v : \tau$ (writing the value v of type τ to address a), and the void call. A right-hand side r is one of an address computation (`addr_of_symbol`, `addr_index`, `addr_field`), a typed load $\text{load } a : \tau$, a typed binary or comparison operator, a unary operator, a typed cast, a value-producing call, or a typed constant. (The grammar of [Appendix D](#) also admits increment/decrement forms, which the lowering never emits and the interpreter therefore never evaluates.) The terminators are the conditional branch $\text{br } c, \ell_t, \ell_f$, the unconditional jump $\text{jmp } \ell$, and the return $\text{ret } [v]$. Every operand is a *value* — either a temporary tn or a typed constant $\#k : \tau$.

Two structural facts matter throughout. First, temporaries are single-assignment *within* a block — each tn is the destination of at most one assignment along any straight-line run through a block — but they are *not* globally in SSA form: the same name may be re-bound in different blocks, and the IR has no ϕ -nodes. Second, all memory access is explicit and typed: a value reaches memory only through a `store` and returns only through a `load`, each carrying the type at which the access is performed, and addresses are computed from frame slots and globals by the `addr_*` family. There is no implicit aliasing of temporaries with memory.

Convention G.1 (Well-formed programs). We write $\vdash P \text{ ok}$ when P satisfies the well-formedness conditions of [Appendix D](#): every function has a non-empty block list whose first block is the entry; every block ends in exactly one terminator and contains no terminator before its end; every label mentioned by a `br`, `jmp` is the label of a block in the same function; every `call` names a function of the program with a matching arity; every operand temporary is dominated by an assignment to it on every path from the entry that reaches the use; and every instruction type-checks under [Appendix D](#)’s typing rules, so that the slot, operand, and result types agree. All sixteen passes take a well-formed program as input; *preservation of $\vdash \cdot \text{ ok}$* is one of the standing obligations of [Section G.1](#).

Operational semantics

We give the IR a small-step operational semantics. The authoritative reference for this semantics is the tree-walking interpreter `IrInterpreter` in `cli/src/main/java/hr/fer/ppj/cli/ir/IrInterpreter.java`; the rules below are its mathematical transcription. Where the interpreter and the rules could drift — integer division by zero, array-bounds checks — we note the interpreter’s actual behaviour explicitly.

States. Let $\text{Val} = \mathbb{Z}$ be the set of machine values: the interpreter represents every scalar — `int32`, `char`, `bool`, pointers, and `float` in Q16.16 fixed point — as a machine word, so a value is just an integer. A *frame* $\varphi = (\rho, \beta)$ pairs a finite *temporary environment* $\rho : \text{Temp} \rightarrow \text{Val}$ with the base addresses of the frame’s slots; the slot bases together with the global table determine, for each slot or global name, the address at which its storage lives. A *memory* $\mu : \text{Addr} \rightarrow \text{Byte}$ is a finite byte-addressable store shared across all frames (the interpreter

allocates each call’s slots into one global μ). A *frame pointer* $\pi = (f, \ell, i)$ records the current function f , the current block label ℓ , and the program point i within that block — i ranging over the instruction indices and a distinguished terminator position. A *call stack* κ is a list of *return contexts*, each recording the caller’s frame, the caller’s continuation point, and the temporary to receive the callee’s result (for a value call) or nothing (for a void call).

A machine *state* is then

$$\sigma = \langle \pi, \varphi, \mu, \kappa \rangle = \langle (f, \ell, i), (\rho, \beta), \mu, \kappa \rangle.$$

The *initial state* for a program P , written $\text{init}(P)$, enters `main` with no arguments: its frame allocates and zero-initializes `main`’s slots in a fresh μ , its temporary environment is empty, its pointer is $(\text{main}, \ell_0, 0)$ for the entry block ℓ_0 , and its call stack is empty. The interpreter’s `initGlobals` step, which fixes the global addresses and their initializers, is folded into this initial μ .

Evaluating values and right-hand sides. The total function $\llbracket v \rrbracket_\varphi \in \text{Val}$ reads a value under a frame: $\llbracket \text{tn} \rrbracket_\varphi = \rho(\text{tn})$ and $\llbracket \#k : \tau \rrbracket_\varphi = k$ (with the float literal first mapped into Q16.16). The partial function $\llbracket r \rrbracket_{\varphi, \mu} \in \text{Val}$ evaluates a right-hand side r : it is the corresponding integer operation for a typed binop or comparison (add as `+`, `cmp_lt` as the 0/1 indicator of $<$, and so on; `mul`, `div` on `float` use the Q16.16 helpers), the address arithmetic for the `addr_*` family, the byte- or word-load $\text{load}_\tau(\mu, \llbracket a \rrbracket_\varphi)$ for load $a : \tau$, the constant for a constant right-hand side, and so on, exactly as `evaluateRhs` computes them.

Convention G.2 (Total evaluation of arithmetic). The interpreter makes every scalar right-hand side *total* by fixing the two C-undefined integer cases: integer `div` and `mod` by zero each yield 0 rather than trapping (`evalBinOp`), and likewise Q16.16 `div` by zero yields 0. The one trapping right-hand side is a checked `addr_index` whose index lies outside its array bound, which raises the bounds trap (code `−6`); the trap aborts the run and is projected to the observation `−6` (see below). All sixteen proofs inherit these conventions and may treat scalar evaluation as a total function on well-typed operands.

Step rules. The small-step relation $\rightsquigarrow$ on states is generated by the rules in Figure G.1, one family per representative instruction or terminator. Writing π^+ for the pointer π advanced to the next program point in the same block, and using the slot/global address function implied by φ for store and address targets:

The auxiliary `enter`($g, \bar{w}, \mu$) allocates and zero-initializes g ’s slots in μ , binds the argument words $\bar{w}$ into g ’s parameter slots, and returns the fresh callee frame together with the grown memory — precisely `allocateSlots` followed by `bindParameters`. A load that targets an aggregate yields the aggregate’s address rather than a word, matching `evaluateRhs`.

Definition G.3 (Multi-step reduction). Let $\rightsquigarrow^*$ be the reflexive-transitive closure of $\rightsquigarrow$: $\sigma \rightsquigarrow^* \sigma'$ iff there is a finite chain $\sigma = \sigma_0 \rightsquigarrow \sigma_1 \rightsquigarrow \dots \rightsquigarrow \sigma_k = \sigma'$ with $k \geq 0$. A state σ is

ASSIGN-OP	$\frac{\pi = (f, \ell, i) \quad I_i = (\text{tn} = r) \quad \llbracket r \rrbracket_{\varphi, \mu} = w}{\langle \pi, \varphi, \mu, \kappa \rangle \rightsquigarrow \langle \pi^+, \varphi[\rho \mapsto \rho[\text{tn} \mapsto w]], \mu, \kappa \rangle}$	
STORE	$\frac{\pi = (f, \ell, i) \quad I_i = (\text{store } a, v : \tau) \quad \mu' = \text{store}_\tau(\mu, \llbracket a \rrbracket_\varphi, \llbracket v \rrbracket_\varphi)}{\langle \pi, \varphi, \mu, \kappa \rangle \rightsquigarrow \langle \pi^+, \varphi, \mu', \kappa \rangle}$	
CALL	$\frac{\pi = (f, \ell, i) \quad I_i = (\text{tn} = \text{call } g(\bar{v}) : \tau) \quad \bar{w} = \llbracket \bar{v} \rrbracket_\varphi \quad (\varphi_g, \mu') = \text{enter}(g, \bar{w}, \mu) \quad \pi_g = (g, \ell_0^g, 0) \quad \xi = \text{ret-ctx}(\pi^+, \varphi, \text{tn})}{\langle \pi, \varphi, \mu, \kappa \rangle \rightsquigarrow \langle \pi_g, \varphi_g, \mu', \xi :: \kappa \rangle}$	
JMP	$\frac{\pi = (f, \ell, i) \quad I_i^{\text{term}} = (\text{jmp } \ell')}{\langle \pi, \varphi, \mu, \kappa \rangle \rightsquigarrow \langle (f, \ell', 0), \varphi, \mu, \kappa \rangle}$	
BR-T	$\frac{I_i^{\text{term}} = (\text{br } c, \ell_t, \ell_f) \quad \llbracket c \rrbracket_\varphi \neq 0}{\langle \pi, \varphi, \mu, \kappa \rangle \rightsquigarrow \langle (f, \ell_t, 0), \varphi, \mu, \kappa \rangle}$	$\text{BR-F} \quad \frac{I_i^{\text{term}} = (\text{br } c, \ell_t, \ell_f) \quad \llbracket c \rrbracket_\varphi = 0}{\langle \pi, \varphi, \mu, \kappa \rangle \rightsquigarrow \langle (f, \ell_f, 0), \varphi, \mu, \kappa \rangle}$
RET-CALLER	$\frac{I_i^{\text{term}} = (\text{ret } v) \quad \llbracket v \rrbracket_\varphi = w \quad \kappa = \xi :: \kappa' \quad \xi = (\pi_c, \varphi_c, \text{tm})}{\langle \pi, \varphi, \mu, \kappa \rangle \rightsquigarrow \langle \pi_c, \varphi_c[\rho_c \mapsto \rho_c[\text{tm} \mapsto w]], \mu, \kappa' \rangle}$	

Figure G.1: Small-step rules for the representative IR instruction and terminator forms. A load is the ASSIGN-OP instance whose r is $\text{load } a : \tau$; a constant assignment is the instance whose r is a constant; a void call drops the tn field of the return context. The RET rule shown returns into a caller; the top-level case, in which κ is empty, is handled by the observation function below.

terminal if it has no $\rightsquigarrow$ -successor; by inspection of Figure G.1 the terminal states are exactly the `ret` configurations (with or without a value) with empty call stack and the bounds-trap configuration.

Remark G.4 (Determinism). For every non-terminal σ there is exactly one σ' with $\sigma \rightsquigarrow \sigma'$. Each rule of Figure G.1 is selected by the instruction or terminator at the current program point, the side conditions of BR-T/BR-F partition on the single test $\llbracket c \rrbracket_\varphi \neq 0$, and every auxiliary ($\llbracket \cdot \rrbracket$, `storer`, `enter`) is a function. The interpreter is correspondingly a deterministic loop. We use determinism freely: it lets the simulation arguments below pick “the” successor of a state without further justification.

The observation function

FRISCCc’s IR subset has no input and no I/O instruction — no read, no write, no volatile memory — so the only thing an external observer of a whole-program run can detect is the value the program ultimately produces. On the FRISC target that is the word the program leaves in the return-value register R6; in the semantics above it is the value of the `ret` v that unwinds `main`’s frame with an empty call stack. We make this the sole observable.

Definition G.5 (Observation). The *observation* of a program P , written $\text{obs}(P) \in \mathbb{Z} \cup \{\perp\}$, is defined by running P from its initial state:

$$\text{obs}(P) = \begin{cases} w & \text{if } \text{init}(P) \rightsquigarrow^* \langle (\text{main}, \ell, i), \varphi, \mu, \varepsilon \rangle \text{ terminal on ret } v \text{ with } \llbracket v \rrbracket_\varphi = w, \\ -6 & \text{if } \text{init}(P) \rightsquigarrow^* \sigma_{\text{trap}} \text{ (the bounds trap of Theorem G.2),} \\ \perp & \text{if the run from } \text{init}(P) \text{ does not terminate.} \end{cases}$$

A bare `ret` out of `main` (no value) observes 0, as in `executeFunction`. Because $\rightsquigarrow$ is deterministic (Theorem G.4), exactly one of the three cases holds, so obs is a well-defined total function on programs into $\mathbb{Z} \cup \{\perp\}$, with $\perp$ recording divergence.

The observation is deliberately coarse. It hides the temporary environment, the layout and contents of the frame memory, the number of steps taken, the sequence of blocks visited, and the call stack. A pass is free to change all of these. What it may not change is obs .

Semantic preservation, defined

Definition G.6 (Pass; semantics preservation). A *pass* is a partial function $\mathcal{P}$ on IR programs, defined at least on every well-formed program. $\mathcal{P}$ is *semantics-preserving* iff for every well-formed program P (Theorem G.1),

$$\text{obs}(\mathcal{P}(P)) = \text{obs}(P).$$

Equality is taken in $\mathbb{Z} \cup \{\perp\}$, so a preserving pass must turn a terminating run into a terminating run with the same value, a trapping run into the same trap, and a diverging run into a diverging run.

Theorem G.6 is the statement each per-pass proof ultimately establishes. Proving it directly — by reasoning about whole runs — is awkward, because a pass typically changes the program *locally* (one instruction, one block, one loop) while the runs are global. The standard remedy, used by every proof that follows, is to exhibit a *simulation relation* between the runs of P and the runs of $\mathcal{P}(P)$ and to check that it is preserved by single steps. We set up that machinery once.

Definition G.7 (Simulation relation). Let P be well-formed and $P' = \mathcal{P}(P)$. A relation $\approx \subseteq \text{State}_P \times \text{State}_{P'}$ between states of the two programs is a *simulation* (for the observation obs) if:

1. **Initial states are related:** $\text{init}(P) \approx \text{init}(P')$.
2. **Steps preserve relation (lock-step or stutter):** whenever $\sigma \approx \sigma'$ and $\sigma \rightsquigarrow \tau$, there is a state τ' with $\sigma' \rightsquigarrow^* \tau'$ and $\tau \approx \tau'$.
3. **Related terminals agree on observation:** whenever $\sigma \approx \sigma'$ and σ is terminal, σ' reduces to a terminal σ'' with the same observation — both return the same word, or both trap.

Clause (2) permits *stuttering* on the P' side: a single source step may be matched by zero, one, or several target steps. This is exactly the freedom a pass needs — deleting a dead assignment matches a source step with *zero* target steps, while inlining matches one source call with *many* target steps. The relation is a “one source step to many target steps” simulation, the plus/star simulation of CompCert’s backward-simulation toolkit [56]. Several passes in fact prove the symmetric statement (a target step matched by source steps); since $\rightsquigarrow$ is deterministic on both sides (**Theorem G.4**), the two directions coincide on terminating runs, and the lemma below is stated so that exhibiting either suffices.

Lemma G.8 (Simulation implies observational equivalence). Let P be well-formed, $P' = \mathcal{P}(P)$, and suppose $\approx$ is a simulation in the sense of **Theorem G.7** equipped with an anti-stuttering measure $\mathfrak{m}$ in the sense of **Theorem G.9**: a function from related pairs into a well-founded order that strictly decreases on every stuttering source step. Then P and P' are observationally equivalent, by which we mean $\text{obs}(P') = \text{obs}(P)$.

Proof. We treat the three possibilities for $\text{obs}(P)$ in turn.

Case $\text{obs}(P) = w \in \mathbb{Z}$ (terminating run). Then $\text{init}(P) \rightsquigarrow^* \sigma_n$ with σ_n terminal and observation w ; write the run as $\sigma_0 \rightsquigarrow \sigma_1 \rightsquigarrow \dots \rightsquigarrow \sigma_n$ with $\sigma_0 = \text{init}(P)$. By clause (1), $\sigma_0 \approx \sigma'_0$ for $\sigma'_0 = \text{init}(P')$. We build, by induction on $k = 0, 1, \dots, n$, target states σ'_k with $\sigma'_0 \rightsquigarrow^* \sigma'_k$ and $\sigma_k \approx \sigma'_k$. The base case is clause (1). For the step, given $\sigma_k \approx \sigma'_k$ and $\sigma_k \rightsquigarrow \sigma_{k+1}$ (which exists for $k < n$ since σ_k is non-terminal), clause (2) yields σ'_{k+1} with $\sigma'_k \rightsquigarrow^* \sigma'_{k+1}$ and $\sigma_{k+1} \approx \sigma'_{k+1}$; composing $\sigma'_0 \rightsquigarrow^* \sigma'_k \rightsquigarrow^* \sigma'_{k+1}$ closes the induction. At $k = n$ we have $\sigma'_0 \rightsquigarrow^* \sigma'_n$ with $\sigma_n \approx \sigma'_n$ and σ_n terminal, so by clause (3) σ'_n reduces to a terminal with observation w . Hence $\text{init}(P') \rightsquigarrow^*$ a terminal observing w , and by determinism (**Theorem G.4**) that terminal is the unique one the run of P' reaches; therefore $\text{obs}(P') = w$.

Case $\text{obs}(P) = -6$ (*trapping run*). Identical to the terminating case: the trap configuration σ_{trap} is terminal ([Theorem G.3](#)), so the same induction reaches a related target state, and clause (3) forces the matching target terminal to be a trap, observed as -6 .

Case $\text{obs}(P) = \perp$ (*diverging run*). Suppose toward a contradiction that $\text{obs}(P') \neq \perp$, i.e. the run of P' terminates at some τ' after m target steps. The source run is an infinite chain $\sigma_0 \rightsquigarrow \sigma_1 \rightsquigarrow \dots$. Running the same inductive construction as above produces, for each k , a target state σ'_k with $\sigma'_0 \rightsquigarrow^* \sigma'_k$ and $\sigma_k \approx \sigma'_k$. Two subcases. If infinitely many of the matching target multi-steps $\sigma'_k \rightsquigarrow^* \sigma'_{k+1}$ are non-empty, the target run is itself infinite, contradicting termination at τ' . Otherwise all but finitely many matching steps are empty stutters; then from some index K onward $\sigma'_K = \sigma'_{K+1} = \dots$ is a fixed target state related to every σ_k for $k \geq K$. But the source makes genuine progress $\sigma_K \rightsquigarrow \sigma_{K+1} \rightsquigarrow \dots$ forever while the target stutters in place; clause (2) applied at each such source step would have to advance the target by a non-empty multi-step infinitely often *or* keep relating the fixed σ'_K to ever-changing source states. The latter is the *infinite-stutter* pathology, and it is ruled out by the measure $\mathbf{m}$ assumed in the hypotheses: along the empty-stutter tail $\sigma_K \rightsquigarrow \sigma_{K+1} \rightsquigarrow \dots$ the target stays fixed at σ'_K , so each step is a pure stutter and $\mathbf{m}(\sigma_{k+1}, \sigma'_K) < \mathbf{m}(\sigma_k, \sigma'_K)$ holds for all $k \geq K$. That is an infinite strictly decreasing chain in a well-founded order, which cannot exist. Hence the run of empty stutters is finite, forcing a non-empty target multi-step, and we are back in the first subcase — contradiction. Therefore $\text{obs}(P') = \perp = \text{obs}(P)$. $\square$

Convention G.9 (Anti-stuttering measure). To make the divergence case of [Theorem G.8](#) go through, each per-pass simulation comes equipped with a function $\mathbf{m}$ from related pairs into a well-founded order (typically $\mathbb{N}$) such that whenever $\sigma \approx \sigma'$, $\sigma \rightsquigarrow \tau$, and the matching target multi-step $\sigma' \rightsquigarrow^* \tau'$ is *empty* (a pure stutter), we have $\mathbf{m}(\tau, \tau') < \mathbf{m}(\sigma, \sigma')$. For most passes $\mathbf{m}$ counts the source instructions the target has not yet “caught up” on, and is bounded by the size of one block, so the condition is immediate. A proof that introduces *no* stuttering at all (true lock-step) may take $\mathbf{m} \equiv 0$ and discharge this convention vacuously.

With [Theorem G.8](#) in hand, the burden on each per-pass proof collapses to a local task.

Corollary G.10 (Proof obligation for a single pass). *To prove that a pass $\mathcal{P}$ is semantics-preserving ([Theorem G.6](#)), it suffices, for an arbitrary well-formed P with $P' = \mathcal{P}(P)$, to:*

1. *exhibit a relation $\approx$ between states of P and P' , together with an anti-stuttering measure ([Theorem G.9](#));*
2. *check the three clauses of [Theorem G.7](#) — which reduces, by [Theorem G.4](#), to a case analysis over the instruction and terminator forms the pass rewrites, leaving the unrewritten forms to a trivial “identity step matches identity step” argument.*

Proof. Immediate from [Theorem G.8](#) and [Theorem G.6](#): the lemma turns a simulation into $\text{obs}(P') = \text{obs}(P)$, which is precisely the definition of preservation. $\square$

In practice a per-pass proof states its $\approx$ in one or two lines (“related states agree on μ and on every temporary the pass did not touch”), observes that every instruction outside the pass’s rewrite set steps in lock-step on both sides under that relation, and spends its real effort on the handful of rewritten forms.

Common obligations

Two further properties are required of *every* pass and are established once here, so the individual proofs may cite this subsection instead of re-deriving them.

Proposition G.11 (Preservation of well-formedness). *If $\vdash P \text{ ok}$ and $P' = \mathcal{P}(P)$ for any of the sixteen passes, then $\vdash P' \text{ ok}$.*

Proof obligation. This is not a theorem about obs but a structural invariant of each pass, and is discharged inside each per-pass proof by checking that the rewrite respects the conditions of [Theorem G.1](#):

- *Control-flow integrity.* Any pass that adds, removes, merges, or redirects blocks must leave every `br/jmp` target pointing at a block of the same function and must keep each block ending in exactly one terminator. (The control-flow and unreachable-block passes carry the bulk of this obligation; most passes touch no terminator and discharge it trivially.)
- *Definition dominance.* Any pass that moves, deletes, or introduces an assignment must preserve the property that every use of a temporary is dominated by a definition of it — the invariant that makes $\llbracket \text{tn} \rrbracket_\varphi$ defined whenever it is read. Code-motion and elimination passes carry this obligation explicitly.
- *Type agreement.* Any pass that rewrites a right-hand side, a store, or a cast must keep operand, slot, and result types in agreement under [Appendix D](#)’s typing rules, so the rewritten instruction still type-checks. Constant-folding and the `int32/shift/cast` passes carry this obligation explicitly.

Because $\approx$ is only required to relate states of two well-formed programs, $\vdash P' \text{ ok}$ is a precondition for invoking [Theorem G.8](#), and so each proof discharges it before exhibiting its simulation. $\square$

Proposition G.12 (Preservation of termination behaviour). *A semantics-preserving pass preserves the termination behaviour of P : P terminates iff P' terminates, and they trap on the same runs.*

Proof. This is a consequence of [Theorem G.6](#), not a separate obligation. Since $\text{obs}(P') = \text{obs}(P)$ in $\mathbb{Z} \cup \{\perp\}$, we have $\text{obs}(P) = \perp$ iff $\text{obs}(P') = \perp$; by [Theorem G.5](#) that is exactly “ P diverges iff P' diverges.” Equality on the $\mathbb{Z}$ part likewise forces the trap value -6 to be

matched by the trap value -6 . We record it separately only because readers sometimes expect termination to be argued on its own; here it is already folded into the $\perp$ case of the observation and into the divergence case of [Theorem G.8](#). $\square$

Finally, a note on *composition*, since the optimizer applies the sixteen passes in sequence and iterates to a fixed point.

Corollary G.13 (Pipelines preserve semantics). *If $\mathcal{P}_1, \dots, \mathcal{P}_k$ are each semantics-preserving ([Theorem G.6](#)) and each preserves well-formedness ([Theorem G.11](#)), then their composition $\mathcal{P}_k \circ \dots \circ \mathcal{P}_1$ is semantics-preserving on well-formed inputs, as is any finite number of iterations of the pipeline.*

Proof. By induction on k . For $k = 1$ the claim is the hypothesis. For the step, let P be well-formed and write $Q = (\mathcal{P}_{k-1} \circ \dots \circ \mathcal{P}_1)(P)$. By the induction hypothesis (using [Theorem G.11](#) at each stage to keep the input well-formed) we have $\text{obs}(Q) = \text{obs}(P)$ and $\vdash Q \text{ ok}$. Since $\mathcal{P}_k$ is preserving and defined on the well-formed Q , $\text{obs}(\mathcal{P}_k(Q)) = \text{obs}(Q) = \text{obs}(P)$. Iteration is the special case in which the $\mathcal{P}_i$ repeat; well-formedness is maintained across iterations by [Theorem G.11](#), so the same argument applies to any finite iteration count. ([Chapter 14](#) reports that the pipeline reaches a fixed point within a few sweeps on the benchmark suite; correctness, however, does not depend on reaching one.) $\square$

This is what every proof in the remainder of the appendix relies on: fix $\approx$, discharge the three simulation clauses by cases on the rewritten forms, confirm the three well-formedness sub-obligations of [Theorem G.11](#), and appeal to [Theorem G.8](#) for the conclusion. The reader who has internalized [Theorem G.10](#) can read each subsequent proof as a single question: *what does this pass change, and why does the observation not notice?*

G.2 Determinism of LR(1) parsing

Chapter 4 builds the canonical collection of LR(1) item sets, derives the ACTION and GOTO tables ([Section 4.6](#)), and runs the shift/reduce driver on a sample input ([Section 4.7](#)). The chapter states without proof that the resulting parser is deterministic: at every step the action it takes is forced by the state on top of its stack and the current lookahead. We collect the proof here. The argument is due, in essence, to Knuth [\[48\]](#); modern textbook presentations are in Aho et al. [\[2\]](#) and Grune and Jacobs [\[33\]](#).

Setup

Fix a context-free grammar $G = (N, T, P, S)$ and the canonical LR(1) parser constructed from it as in [Section 4.4](#) and [Section 4.6](#). Let Q be the set of LR(1) states. We treat $\text{ACTION} : Q \times (T \cup \{\$\}) \rightarrow \mathcal{A}$ and $\text{GOTO} : Q \times N \rightarrow Q \cup \{\perp\}$ as *set-valued* maps during

construction, where $\mathcal{A}$ contains shift, reduce, accept, and error entries; the *LR(1) hypothesis* is that every ACTION cell of G 's canonical table contains at most one non-error action.

A parser configuration is a pair (S, w) where $S = q_0 q_1 \cdots q_n \in Q^+$ is the state stack (bottom to top) and $w \in T^* \cdot \{\$$ is the remaining input with end marker. The initial configuration on input u is $(q_0, u\$)$. The parser's small-step transition relation $\leadsto$ is defined by cases on $a := \text{ACTION}(q_n, \text{head}(w))$, where $\text{head}(w)$ is the first symbol of w :

shift q' : $(S, aw') \leadsto (Sq', w')$.

reduce $A \rightarrow \alpha$: with $|\alpha| = k$, write $S = S' q_{n-k+1} \cdots q_n$; let $q'' = \text{GOTO}(\text{top}(S'), A)$. Then $(S, w) \leadsto (S' q'', w)$.

accept: the configuration is terminal and the parser succeeds.

error: the configuration is terminal and the parser rejects.

The driver of Section 4.7 implements exactly this relation. The two lemmas below establish, in turn, that a single step is uniquely determined and that a full run is finite.

Single-step determinism

Lemma G.14 (Single-valued step). *If G is LR(1), then for every non-terminal configuration (S, w) there is exactly one configuration (S', w') with $(S, w) \leadsto (S', w')$, or the configuration is terminal.*

Proof. The next-step choice is driven by $a = \text{ACTION}(\text{top}(S), \text{head}(w))$. By the LR(1) hypothesis, the cell holds at most one non-error action. The shift case fixes the destination state q' from the same cell, so the new stack and new remaining input are determined. The reduce case fixes the production $A \rightarrow \alpha$, hence $k = |\alpha|$, hence the prefix of S to pop; the GOTO table is single-valued by construction (it is a function from $Q \times N$, defined during the canonical collection), so the new top is determined. Accept and error yield terminal configurations and produce no successor. In every case, the configuration update is uniquely determined. $\square$

Termination

Lemma G.15 (Termination). *On input of length n , the parser performs at most $O(n)$ steps before reaching a terminal configuration.*

Proof. Every shift consumes one input symbol, so the number of shifts is at most $n + 1$ (counting the end marker). It remains to bound the number of consecutive reductions between two shifts.

Assign to a configuration (S, w) the height $h(S)$ of the *symbol stack*: the number of grammar symbols (terminals and non-terminals) the parser has shifted or reduced but not yet collapsed, which equals the number of states on S above q_0 . A reduce by $A \rightarrow \alpha$ with $|\alpha| = k \geq 1$ pops k states and pushes one, so $h(S)$ strictly decreases; a shift increases $h(S)$ by one while consuming an input symbol.

For reductions by a non-empty production this already bounds the consecutive reductions between two shifts: each strictly lowers $h(S)$, which is non-negative, so no more than $h(S)$ of them can occur in a row. The ϵ -production case $|\alpha| = 0$ escapes this measure, because reducing by $A \rightarrow \epsilon$ pops nothing and pushes one state, so $h(S)$ *increases*. Here we do not attempt a self-contained descent; instead we invoke the classical linear-time bound for canonical LR parsing, established by Knuth [48] and restated in Aho et al. [2, §4.7]: a deterministic LR parser performs $O(n)$ moves in total on an input of length n , because the finiteness of the canonical collection prevents any unbounded chain of reductions — including empty ones — without an intervening shift. The total step count is therefore linear in n . $\square$

The determinism theorem

Theorem G.16 (Determinism of LR(1) parsing). *Let G be an LR(1) grammar. For every input string $u \in T^*$ the LR(1) parser of Section 4.7 either*

1. *accepts u , producing a unique parse tree whose yield is u ; or*
2. *rejects u at a uniquely determined position.*

The action taken at every step is fully determined by the top of the state stack and the current lookahead; the parser makes no nondeterministic choices at runtime.

Proof. We argue by induction on the step count k that the configuration (S_k, w_k) reached from $(q_0, u\$)$ after k steps is uniquely determined.

Base case ($k = 0$). The initial configuration $(q_0, u\$)$ depends only on the input u and on the grammar's start state q_0 , both fixed.

Inductive step. Suppose (S_k, w_k) is uniquely determined. If it is a terminal configuration the run halts and (S_{k+1}, w_{k+1}) does not exist; otherwise Theorem G.14 delivers a unique successor (S_{k+1}, w_{k+1}) .

By Theorem G.15 the chain of configurations is finite, so there is a least K at which (S_K, w_K) is terminal. The inductive argument shows that K and the terminal configuration are both functions of u alone. If the terminal action is accept, the parse tree assembled bottom-up by the reductions along the run is likewise a function of u , because the reductions themselves are fixed by the run. If the terminal action is error, the rejecting step K is uniquely determined.

This establishes both clauses of the theorem. $\square$

Unambiguity as a corollary

Corollary G.17 (Every LR(1) grammar is unambiguous). *If G is LR(1) and $u \in L(G)$, then u has exactly one parse tree in G .*

Proof. Suppose for contradiction that $u \in L(G)$ admits two distinct parse trees $\tau_1 \neq \tau_2$. A standard property of LR parsing is that its sequence of reductions reproduces, in reverse, a *rightmost* derivation of u ; this is the content of the handle-finding argument in Section 4.4. The handle of a right-sentential form is uniquely determined by the form itself: it is the leftmost complete right-hand side appearing as a substring after any reduction. Reading τ_1 and τ_2 as rightmost derivations of u , both derivations must therefore agree on every reduction step, hence $\tau_1 = \tau_2$, contradicting our assumption. $\square$

Remark G.18 (The converse fails). The implication $\text{LR}(1) \Rightarrow \text{unambiguous}$ is strict. Aho et al. [2, §4.7] give the canonical counterexample: the grammar generating palindromes $S \rightarrow aSa \mid bSb \mid \epsilon$ is unambiguous, but no fixed lookahead suffices to decide a handle in its middle, so it is not $\text{LR}(k)$ for any k . The LR(1) class is nonetheless wide enough to cover every mainstream programming language; the FRISCCc grammar of Section 4.9 is one instance.

G.3 Pass: typed constant folding

What the pass does

Typed constant folding is the most local optimization in the pipeline: it evaluates, at compile time, any right-hand side all of whose operands are already typed constants, and replaces the right-hand side with the single constant it computes. The implementation is `TypedConstantFoldingPass` in `compiler-opt`. It walks every block of every function and, for each assignment $tn = r$, inspects r . A binary operator, comparison, unary operator, or cast whose operands (`constOf`) are all literals is handed to `foldBinOp`, `foldCmp`, `foldUnary`, or `foldCast`; if that helper returns a constant, the right-hand side becomes a `ConstRhs` wrapping it. Every other right-hand side, and every right-hand side with a non-constant operand, is left untouched. Terminators are inspected too, but `foldValue` is the identity on values, so a `br` or `ret` is only ever “rewritten” to itself.

The two facts that make the pass delicate, rather than trivial, are *type fidelity* and *the float guard*. The folded constant must carry the same static type the original right-hand side produced, because the rest of the IR was type-checked against that type; `normalizeIntConst` enforces this by re-truncating the host int result into the declared width (bool to 0/1, char and uchar to a byte, int32 to a full 32-bit word). And float folding commits only when `isRoundTripStable` confirms that the computed Q16.16 raw word survives a $\text{float} \rightarrow \text{raw} \rightarrow \text{float}$ round trip; when it does not, `foldBinOp` / `foldUnary` / `foldCast` return null and the

original right-hand side is kept. We will see that both choices are exactly what the proof needs.

Example G.19 (A folded comparison). The block fragment `t3 = cmp_lt #3:int32, #5:int32` becomes `t3 = #1:bool`, because `foldCmp` evaluates $3 < 5$ to `true` and `normalizeIntConst` packages it as the `bool` constant 1. The result type is `bool` both before (the comparison’s result type) and after (the constant’s type): it survives the rewrite, as required.

The claim

Theorem G.20 (Semantic preservation of typed constant folding). *Let $\mathcal{P}_{\text{tcf}}$ be the typed-constant-folding pass. For every well-formed program P (Theorem G.1), $\text{obs}(\mathcal{P}_{\text{tcf}}(P)) = \text{obs}(P)$.*

Proof

We discharge the obligation set out in Theorem G.10: fix a simulation, check the rewritten forms step in lock-step, and confirm the well-formedness sub-obligations of Theorem G.11.

The pass changes neither the control-flow graph nor the set of temporaries: every block of $P' = \mathcal{P}_{\text{tcf}}(P)$ has the same label, the same number of instructions in the same order with the same destinations, and the same terminator as the corresponding block of P . Only the *right-hand sides* of some assignments differ, and each differing pair (r, r') has $r' = \#k$ a constant while r is a binop/cmp/unary/cast over constant operands. This makes the simulation an exact lock-step identity on states.

Definition G.21 (Simulation for $\mathcal{P}_{\text{tcf}}$). Relate $\sigma \approx \sigma'$ iff σ and σ' are *equal* as states — identical pointer (same f, ℓ, i), identical frame (ρ, β) , identical memory μ , identical call stack κ — where the pointer of σ' is read against P' ’s block list and the pointer of σ against P ’s. Take the anti-stuttering measure $\mathbf{m} \equiv 0$ (Theorem G.9); the simulation is pure lock-step with no stutter.

Initial states are related because `init` depends only on `main`’s slot table and entry label, which the pass leaves unchanged; so clause (1) of Theorem G.7 holds. For clause (3): related terminal states are *identical* states, so they trivially share an observation. Everything therefore rests on clause (2), the single-step case. Since the program point, the instruction shapes, and the terminators coincide, the only rule whose two sides could diverge is `ASSIGN-OP` at an assignment whose right-hand side the pass folded. We isolate that case in one lemma.

Lemma G.22 (Folding preserves the evaluated value). *Let r be a right-hand side that the pass replaces by the constant $r' = \#k : \tau$, where τ is the static result type of r . Then for every frame φ and memory μ , $\llbracket r \rrbracket_{\varphi, \mu} = \llbracket r' \rrbracket_{\varphi, \mu} = k$.*

Proof. Because the pass folds r only when `constOf` reports all operands as literals, $\llbracket r \rrbracket_{\varphi, \mu}$ does not depend on φ or μ : it is the closed value the operational semantics assigns to a right-hand side over constant operands. We check that the folding helper computes exactly that value, by the four cases of `foldRhs`.

Integer binop (`foldBinOp`, $\tau \neq \text{float}$). The helper folds all ten `BinOpName` operators on the two host-int words and then applies `normalizeIntConst`. The arithmetic cases (`add`, `sub`, `mul`) use host $+$, $-$, $\times$, and `div/mod` use the wraparound-correct `Int32Semantics.divide/modulo` with the divisor-zero result fixed to 0. The bitwise cases (`and`, `or`, `xor`) use host $\&$, $|$, $\wedge$ on the 32-bit words, and the shift cases (`shl`, `shr`) use host $\ll$ and $\gg$ — which, following the JVM, mask the shift count to its low five bits and perform an arithmetic (sign-propagating) right shift. The operational semantics of Figure G.1, realised by `evalBinOp` in the interpreter, evaluates the same operator on the same two integer words with the *identical* host operators ($\&$, $|$, $\wedge$, $\ll$, $\gg$, and the same count-masking on the shifts), so the unnormalised words coincide. Its result for `div/mod` by zero is 0 by Theorem G.2, matching the helper; and its result is read back at width τ , which is precisely what `normalizeIntConst` does (mask to a byte for `char/uchar`, clamp to 0/1 for `bool`, keep the full word for `int32`). Hence in every one of the ten cases the folded k equals $\llbracket r \rrbracket_{\varphi, \mu}$.

Comparison (`foldCmp`). `evaluateComparison` returns the boolean of the relation, which the pass packages as the `bool` constant 1 or 0 — exactly the 0/1 indicator that $\llbracket \text{cmp_}* \rrbracket_{\varphi, \mu}$ produces.

Integer unary (`foldUnary`, $\tau \neq \text{float}$). `NEG` $\mapsto -x$, `NOT` $\mapsto [x = 0]$, `BITNOT` $\mapsto \neg x$, again re-normalized to width τ , matching the semantics.

Float binop / unary / cast ($\tau = \text{float}$, or a cast). The helper computes in Q16.16 raw words via `Q16FloatSemantics`, the same fixed-point routines the interpreter’s four float binops (`add/sub/mul/div-on-float`) and the `itof/ftoi` casts use in `evaluateRhs`. *Crucially*, the pass commits the folded float constant only when `isRoundTripStable` holds for the raw word; otherwise it returns null and no rewrite occurs. So in every case where a float rewrite *is* performed, the stored `FloatConst` re-encodes to the very raw word the computation produced, and $\llbracket r' \rrbracket_{\varphi, \mu}$ (which re-derives that raw word from the literal) equals $\llbracket r \rrbracket_{\varphi, \mu}$. When the round trip fails, r is unchanged and the case does not arise.

For the cast cases (`foldCast`), the helper applies the same bit-level operation the semantics assigns to each cast kind — `TRUNC/ZEXT` mask to a byte, `SEXT` sign-extends from bit 7, `ITOF` shifts left 16, `FTOI` shifts right 16, `PTRCAST` is the identity — and then normalizes to the declared result type, including the null-pointer special case when the result type is a pointer and the word is 0. In each sub-case the produced k matches $\llbracket r \rrbracket_{\varphi, \mu}$.

In all cases $\llbracket r \rrbracket_{\varphi, \mu} = k = \llbracket \#k : \tau \rrbracket_{\varphi, \mu}$. □

Proof of Theorem G.20, concluded. Take $\sigma \approx \sigma'$ (so $\sigma = \sigma'$ as states) and $\sigma \rightsquigarrow \tau$. If the current instruction is not a folded assignment, the same rule fires on both sides over identical operands and produces identical successors, so $\tau \approx \tau$ with the target taking one step. If it *is* a folded assignment `tn = r` on the source and `tn = r'` on the target, `ASSIGN-OP` fires on each;

by [Theorem G.22](#) the bound word $w = \llbracket r \rrbracket_{\varphi, \mu} = \llbracket r' \rrbracket_{\varphi, \mu}$ is the same, the pointer advances identically, and μ, κ are unchanged, so again the two successors are equal and related. Clause (2) holds with a single matching target step ($\mathbf{m} \equiv 0$, no stutter). With clauses (1) and (3) already noted, $\approx$ is a simulation, and [Theorem G.8](#) gives $\text{obs}(P') = \text{obs}(P)$.

Well-formedness ([Theorem G.11](#)). Control-flow integrity and definition dominance are immediate: the CFG, the block structure, every terminator target, and every assignment destination are untouched, so no use loses its dominating definition. Type agreement is the substance of the pass and is preserved because [Theorem G.22](#)'s constant carries the result type τ of the right-hand side it replaces (the float guard never lets a non-representable result masquerade as a float literal). Hence $\vdash P' \text{ ok}$. $\square$

Implementation pointers

The folding helpers are `foldBinOp`, `foldCmp`, `foldUnary`, `foldCast`, and the width-restoring `normalizeIntConst`, all in `TypedConstantFoldingPass`. The fixed-point routines and the round-trip guard live in `Q16FloatSemantics` (`addRaw/subRaw/mulRaw/divRaw`, `isRoundTripStable`); the wraparound-correct integer division and modulo live in `Int32Semantics`.

Two honest notes on where the implementation is narrower than the theorem might suggest. First, the pass is *conservative on floats*: whenever `isRoundTripStable` fails it declines to fold, so it can leave a foldable-looking float expression in place. This never threatens preservation (an un-folded expression is trivially equivalent to itself); it only forgoes an opportunity. Second, divide/modulo by a zero constant fold to 0 rather than to a trap, matching [Theorem G.2](#) and the interpreter's `evalBinOp`; on the FRISC target the same expression also yields 0 through the `F_DIV/F_MOD` helpers, so the constant the pass installs agrees with the run-time behaviour it replaces.

G.4 Pass: int32 algebraic simplification

What the pass does

Where typed constant folding ([Section G.3](#)) needs *both* operands constant, the int32 arithmetic pass exploits algebraic identities that hold when only *one* operand is a known constant, or when the two operands are syntactically equal. The implementation is `Int32ArithmeticPass` in `compiler-opt`. It fires only on right-hand sides whose result type is exactly `int32` (`simplifyBinOp` bails out otherwise), so the proof never has to reason about `char`, `bool`, or float width effects. Within `int32`, `simplifyBinOp` applies the following rewrites,

reading x for an arbitrary operand value and c for an `int32` literal:

$x + \#0 \rightsquigarrow x,$	$\#0 + x \rightsquigarrow x,$	$x - \#0 \rightsquigarrow x,$
$x - x \rightsquigarrow \#0,$	$x \times \#0 \rightsquigarrow \#0,$	$\#0 \times x \rightsquigarrow \#0,$
$x \times \#1 \rightsquigarrow x,$	$\#1 \times x \rightsquigarrow x,$	$x \times \#(-1) \rightsquigarrow -x,$
$\#(-1) \times x \rightsquigarrow -x,$	$x \div \#1 \rightsquigarrow x,$	$x \div \#(-1) \rightsquigarrow -x,$
$x \bmod \#1 \rightsquigarrow \#0,$	$x \bmod \#(-1) \rightsquigarrow \#0.$	

It also closes the constant-constant cases of $+$, $-$, $\times$, $\div$, `mod` (overlapping with the folding pass), and `simplifyUnary` collapses a double negation $-(-x) \rightsquigarrow x$ when the inner `NEG` is recorded in the block-local definitions map.

Two implementation choices shape the proof. First, the “simpler operand” is not spliced in as a bare value: `identity(v)` emits the right-hand side $v + \#0 : \text{int32}$ and `neg(v)` emits $-v : \text{int32}$, both still well-typed assignments. Second, the equality test for $x - x$ is the *syntactic* `left.equals(right)`: the two operands must be the same temporary or the same literal, which (because temporaries are single-assignment within a block) means they denote the same value. We must respect 32-bit two’s-complement wraparound throughout, since $\llbracket \cdot \rrbracket$ on `int32` is taken modulo 2^{32} .

Example G.23 (A strength-free identity). The one-line program `int ident(int x) { return x * 1; }` lowers to a multiply by the unit constant. With `t1` the load of the parameter `x`, the front end emits `t2 = mul t1, #1:int32`, which the pass rewrites to the identity `t2 = add t1, #0:int32`. The pass-local rewrite is what the downstream copy-propagation and dead-temp passes then consume: at full `-O1` they fold the `add #0` away entirely. The value of `t2` is `t1` before and after.

The claim

Theorem G.24 (Semantic preservation of `int32` algebraic simplification). *Let $\mathcal{P}_{\text{arith}}$ be the `int32` arithmetic pass. For every well-formed program P , $\text{obs}(\mathcal{P}_{\text{arith}}(P)) = \text{obs}(P)$.*

Proof

As with constant folding, the pass alters only right-hand sides of `int32` assignments; it never moves an instruction, never changes a destination, never touches a terminator or the CFG. The simulation is again the equal-states relation of [Theorem G.21](#) (identical pointer, frame, memory, and call stack; $\mathbf{m} \equiv 0$), and clauses (1) and (3) of [Theorem G.7](#) hold verbatim for the same reasons. The whole content is the `ASSIGN-OP` case at a rewritten right-hand side, which we reduce to one arithmetic lemma. Write mod_{32} for reduction into the signed 32-bit range, the interpretation $\llbracket \cdot \rrbracket$ gives every `int32` operator.

Lemma G.25 (Each identity preserves the evaluated value). *For every rewrite $r \rightsquigarrow r'$ that the pass performs on an int32 right-hand side, and every frame φ and memory μ , $\llbracket r' \rrbracket_{\varphi, \mu} = \llbracket r \rrbracket_{\varphi, \mu}$ in mod_{32} arithmetic.*

Proof. Let $x = \llbracket v \rrbracket_{\varphi}$ be the (single) word denoted by the variable operand. Two's-complement addition, subtraction, and multiplication are the ring operations of $\mathbb{Z}/2^{32}\mathbb{Z}$, so the additive and multiplicative identities and the annihilator hold exactly, with no overflow caveat:

$$x + 0 = x, \quad 0 + x = x, \quad x - 0 = x, \quad x - x = 0, \quad x \cdot 0 = 0 = 0 \cdot x, \quad x \cdot 1 = x = 1 \cdot x.$$

The pass emits $x + 0$ for the first three and the two unit-multiply cases, 0 for the annihilator and for $x - x$ and $x \bmod (\pm 1)$, all of which match the displayed equalities. The case $x - x$ relies on the two operands denoting the *same* word; `left.equals(right)` guarantees they are syntactically identical, so $\llbracket left \rrbracket_{\varphi} = \llbracket right \rrbracket_{\varphi}$ in every frame — hence the difference is 0.

For the negation cases, $x \cdot (-1) \equiv -x \pmod{2^{32}}$ holds in the ring, and the pass emits `NEG x (int32)`, whose semantics is exactly $-x \bmod_{32}$; likewise $-x \cdot (-1) \equiv x$ is covered by `simplifyUnary`'s double-negation collapse, valid because the inner definition is the int32 NEG recorded in the block. The one arithmetic subtlety is division. $x \div 1 = x$ in $\mathbb{Z}$ and the quotient never overflows, so the rewrite is exact. For $x \div (-1)$ the pass emits $-x$; this agrees with `div` because the semantic bracket on int32 division is the interpreter's `evalBinOp`, which computes plain Java two's-complement int division (`left/right`). At the lone overflow point that division yields $\text{INT_MIN} \div (-1) = \text{INT_MIN}$ — exactly the two's-complement value of $-\text{INT_MIN}$, which is the result of `NEG x` — so $-x = x \div (-1)$ even there. The pass's own `Int32Semantics.divide` agrees at the same point, but it is `evalBinOp` that backs `div`. Likewise $x \bmod 1 = 0$, and the interpreter's modulo (`left%right`) gives $x \bmod (-1) = 0$ including at `INT\_MIN` (where $\text{INT_MIN} \% (-1) = 0$ in Java), matching the constant the pass installs.

The constant-constant cases reproduce [Theorem G.22](#)'s integer computation and need no separate argument. $\square$

Proof of Theorem G.24, concluded. Take $\sigma \approx \sigma'$ (equal states) and $\sigma \rightsquigarrow \tau$. If the current instruction is unrewritten, identical rules fire on identical operands and yield equal, related successors. If it is a rewritten int32 assignment `tn = r` versus `tn = r'`, `ASSIGN-OP` fires on each and, by [Theorem G.25](#), binds the same word w to `tn`; the pointer advances identically and μ, κ are unchanged, so the successors are equal and related. Clause (2) holds in lock-step. With clauses (1) and (3) from [Theorem G.21](#), $\approx$ is a simulation and [Theorem G.8](#) yields $\text{obs}(P') = \text{obs}(P)$.

Well-formedness (Theorem G.11). The CFG, block structure, and all destinations are unchanged, so control-flow integrity and definition dominance hold trivially — note in particular that `identity/neg` reuse *existing* operands, introducing no new temporary and so no new use to dominate. Type agreement holds because every replacement right-hand side ($v + \#0 : \text{int32}$, $-v : \text{int32}$, or an int32 constant) has result type int32, the same type the original int32 right-hand side carried. Hence $\vdash P' \text{ ok}$. $\square$

Implementation pointers

The rewrites are in `Int32ArithmeticPass.simplifyBinOp` and `simplifyUnary`; the constructors `identity`, `neg`, `intConst` build the replacement right-hand sides; the wraparound-correct division and modulo are `Int32Semantics.divide` and `modulo`.

Two scope notes. The pass guards on `resultType() == IrPrimitiveType.INT32`, so none of these identities is applied to `char`, `bool`, `float`, or pointer arithmetic; the proof's appeal to exact $\mathbb{Z}/2^{32}\mathbb{Z}$ identities is therefore sound precisely where the pass acts. And the $x - x \rightsquigarrow 0$ rule is intentionally syntactic (`left.equals(right)`); it does *not* attempt to prove two distinct temporaries equal, so it can miss a semantically-zero difference — a missed opportunity, never an unsound rewrite.

G.5 Pass: int32 shift simplification

What the pass does

The shift pass turns multiplications by a power of two into a left shift, and erases shifts by zero. The implementation is `Int32ShiftPass` in `compiler-opt`. Like the arithmetic pass it acts only on right-hand sides whose result type is exactly `int32` (`simplifyBinOp` returns early otherwise), and it touches only `shl`, `shr`, and `mul`. The rewrites are, with x an arbitrary operand and c an `int32` literal:

$$x \ll \#0 \implies x, \quad x \gg \#0 \implies x, \quad x \times \#2^k \implies x \ll \#k \quad (2^k > 1),$$

and symmetrically $\#2^k \times x \implies x \ll \#k$. Here `Int32Semantics.isPowerOfTwo` requires the constant to be a *positive* power of two (it rejects 0, negatives, and `INT_MIN` since the test is $v > 0 \wedge (v \& (v - 1)) = 0$), and `powerOfTwoShiftAmount` returns its trailing-zero count k , so $2^k > 1$ means $1 \leq k \leq 30$. As before, the kept operand is wrapped: `shift-by-zero` produces $x + \#0 : \text{int32}$ via `identity`, and the strength reduction produces a genuine `shl` right-hand side.

The interesting part of the design is what the pass *does not* do. It rewrites only *left* shifts, where the signed/unsigned distinction is invisible: `shl` fills with zeros and $x \cdot 2^k \equiv x \ll k \pmod{2^{32}}$ for every two's-complement x , including negative x and including the cases that overflow. It does *not* rewrite division by a power of two into a right shift, so the proof never has to confront the arithmetic-versus-logical right-shift hazard (where $x \div 2^k$ and $x \gg k$ disagree for negative x because of rounding direction). We return to this divergence from the chapter's informal description in the pointers below.

Example G.26 (Strength reduction of a source-level multiply). Take the program `int scale(int x) { return x * 4; }`. The source-level multiply lowers to `t2 = mul t1, #4:int32` (where `t1` is the load of the parameter `x`), and the pass rewrites it to `t2 = shl t1, #2:int32`. For every 32-bit word in `t1` the two agree modulo 2^{32} .

The claim

Theorem G.27 (Semantic preservation of int32 shift simplification). *Let $\mathcal{P}_{\text{shift}}$ be the int32 shift pass. For every well-formed program P , $\text{obs}(\mathcal{P}_{\text{shift}}(P)) = \text{obs}(P)$.*

Proof

The structural situation is identical to the previous two passes: only int32 assignment right-hand sides change, in place, with the same destinations; the CFG, block list, and terminators are untouched. We reuse the equal-states simulation of Theorem G.21 with $\mathbf{m} \equiv 0$; clauses (1) and (3) of Theorem G.7 hold for the same reasons given there. The work is the one ASSIGN-OP case at a rewritten right-hand side, which the next lemma settles. Write $\ll, \gg$ for the interpreter's 32-bit shift operators and mod_{32} for reduction into the signed 32-bit range.

Lemma G.28 (Each shift rewrite preserves the evaluated value). *For every rewrite $r \Rightarrow r'$ the pass performs, and every frame φ and memory μ , $\llbracket r' \rrbracket_{\varphi, \mu} = \llbracket r \rrbracket_{\varphi, \mu}$.*

Proof. Let $x = \llbracket v \rrbracket_{\varphi}$ be the word denoted by the operand kept by the rewrite.

Shift by zero. For any 32-bit shift, $x \ll 0 = x$ and $x \gg 0 = x$ on the nose — a shift of zero positions moves no bits and selects no fill, regardless of the sign of x or whether $\gg$ is the logical or the arithmetic variant. The pass replaces both by $x + \#0 : \text{int32}$, whose value is $x + 0 = x$ in $\mathbb{Z}/2^{32}\mathbb{Z}$. So $\llbracket r' \rrbracket = x = \llbracket r \rrbracket$.

Multiply by a power of two. The pass fires only when the constant factor is 2^k with $1 \leq k \leq 30$ (positive, a power of two, and greater than one, by `isPowerOfTwo` and the > 1 guard), and emits $x \ll \#k$. In $\mathbb{Z}/2^{32}\mathbb{Z}$, multiplication by 2^k is, by definition of the two's-complement representation, identical to a left shift by k : both compute $x \cdot 2^k \bmod 2^{32}$, shifting the low $32 - k$ bits up and discarding the bits that leave bit 31. This identity is uniform in the sign of x — it needs no non-negativity hypothesis — because `shl` is a logical operation that ignores the sign bit, and `mul` on int32 is itself the truncating product $x \cdot 2^k \bmod_{32}$. Hence $\llbracket x \ll \#k \rrbracket_{\varphi} = x \cdot 2^k \bmod_{32} = \llbracket x \times \#2^k \rrbracket_{\varphi}$. The symmetric case $\#2^k \times x$ is the same by commutativity of $\llbracket \times \rrbracket$.

There is no right-shift rewrite to discharge, so the arithmetic-versus-logical $\gg$ distinction never enters the argument. $\square$

Proof of Theorem G.27, concluded. Take $\sigma \approx \sigma'$ (equal states) with $\sigma \rightsquigarrow \tau$. An unrewritten instruction steps identically on both sides. A rewritten int32 assignment $tn = r$ versus $tn = r'$ fires ASSIGN-OP on each; Theorem G.28 gives the same bound word w , the pointer advances identically, and μ, κ are unchanged, so the successors are equal and related. Clause (2) holds in lock-step, and with clauses (1) and (3) inherited from Theorem G.21, $\approx$ is a simulation; Theorem G.8 gives the conclusion.

Well-formedness (Theorem G.11). The CFG and all destinations are unchanged, discharging control-flow integrity and definition dominance trivially; the replacement operands are operands that already appeared, so no new use is introduced. Type agreement holds because each replacement right-hand side ($x + \#0 : \text{int32}$ or $x \ll \#k : \text{int32}$) has result type `int32`, matching the original. Hence $\vdash P' \text{ ok}$. $\square$

Implementation pointers

The rewrites are `Int32ShiftPass.simplifyBinOp`; the power-of-two predicate and shift amount are `Int32Semantics.isPowerOfTwo` and `powerOfTwoShiftAmount`; the replacement builders are `shiftLeft` and `identity`.

One honest divergence from the chapter’s informal sketch. The narrative in [Chapter 7](#) motivates shift strength reduction in both directions — “multiply becomes left shift, divide becomes right shift.” The implemented pass performs *only* the multiply-to-left-shift half and the shift-by-zero erasure; it never rewrites $x \div 2^k$ into a right shift. This is the safe choice, and deliberately so: for negative x , signed division rounds toward zero whereas an arithmetic right shift $\gg$ rounds toward $-\infty$ (e.g. $(-1) \div 2 = 0$ but $(-1) \gg 1 = -1$), so the naive rewrite is unsound, and a logical shift is wrong for negatives in the other direction. Because the pass omits the rewrite entirely, the subtle case never has to be guarded — and the proof above has nothing to apologize for. Any future addition of a division-to-shift rule would need its own lemma distinguishing the signed (`ashr`, with a bias correction for negative dividends) and unsigned (`lshr`) cases, and a fresh entry in this appendix.

G.6 Pass: cast simplification

What the pass does

Lowering inserts a cast whenever the source program crosses a type boundary, and many of those casts turn out to be identities: a value is “converted” to the type it already has. The cast-simplification pass deletes such casts and threads the original value through to every later use. The implementation is `CastSimplificationPass` in `compiler-opt`. It works one block at a time, maintaining a block-local *alias map* from a temporary index to the value it should be replaced by. Walking the block in order, it first rewrites the current instruction’s operands through the alias map (`resolveAlias`, which chases alias chains with a cycle guard). Then, if the instruction is an assignment $tn = \text{cast } v$ whose cast is *redundant* — `isRedundantCast`, defined as `operand.type().equals(resultType)`, i.e. the source and target static types are identical — it does *not* emit the assignment at all; instead it records $tn \mapsto \text{resolve}(v)$ in the alias map and removes any stale alias that pointed at tn . Any other assignment clears tn from the alias map (a fresh definition shadows an old alias) and is emitted with rewritten operands. The block terminator’s operands are rewritten through the final map as well.

The crucial guard is that a cast is dropped *only* when its operand already has the cast’s result type. So the pass never changes the static type observed by a downstream instruction: tn had the cast’s result type τ , and the value substituted for it, v , also has type τ . Nothing else is folded — there is no trunc-then-zext collapse, no itof/ftoi cancellation; the only “chain folding” is the transitive closure the alias map provides when one identity cast feeds another.

Example G.29 (An identity cast removed). After lowering a char-to-char coercion the IR contains

$$t4 = \text{zext } t3 : \text{char}, \quad \text{store } t9, t4 : \text{char}.$$

If $t3$ already has type `char` the cast is redundant; the pass deletes the assignment to $t4$, records $t4 \mapsto t3$, and rewrites the store to `store t9, t3 : char`. The block is one instruction shorter and every later mention of $t4$ now reads $t3$.

The claim

Theorem G.30 (Semantic preservation of cast simplification). *Let $\mathcal{P}_{\text{cast}}$ be the cast-simplification pass. For every well-formed program P , $\text{obs}(\mathcal{P}_{\text{cast}}(P)) = \text{obs}(P)$.*

Proof

Unlike the three preceding passes, this one *deletes* instructions, so the source and target blocks no longer step in exact lock-step: the source executes an assignment that the target has erased. The simulation must therefore (i) allow the target to *stutter* — match the source’s redundant-cast step with zero target steps — and (ii) account for the deleted temporary, whose binding survives in the source frame but not the target frame. We make both precise with a per-block alias function.

The block alias function. Fix a function f and one of its blocks B , with rewritten counterpart B' . Walking B as the pass does induces, at each program point i within B , an alias map A_i from temporary indices to values. Define the *resolution* $\widehat{A}_i$ on values by chasing A_i to a fixed point (the cycle-guarded `resolveAlias`): $\widehat{A}_i(c) = c$ for a constant, and $\widehat{A}_i(tm)$ is the value reached from tm by following A_i until it lands on a temporary not in the map’s domain or on a constant. The pass guarantees A_i is acyclic on live entries (it removes aliases pointing at a just-redefined temporary before re-inserting), so $\widehat{A}_i$ is well defined.

Definition G.31 (Simulation for $\mathcal{P}_{\text{cast}}$). Relate $\sigma \approx \sigma'$ iff

- the memories agree, $\mu = \mu'$, and the call stacks agree position-for-position under the same relation applied at each return context;

- the pointers point at *corresponding* program points: the same function and block, and the source index i and target index i' are the positions of the same original instruction (the target index lags by the number of redundant casts deleted before it in the block);
- the frames agree on the value of every *live* temporary: for every temporary tm that is read at or after the current point along some path, $\llbracket tm \rrbracket_\varphi = \llbracket \widehat{A}_i(tm) \rrbracket_{\varphi'}$. Equivalently, the source frame may bind extra deleted-cast temporaries that the target frame does not, but every value the target will actually consult equals the source value it stands for.

Take the anti-stuttering measure $\mathbf{m}(\sigma, \sigma')$ to be the number of deleted-cast instructions remaining between the current source point and the end of the block — a quantity bounded by the block length and strictly decreased by each stuttering step (Theorem G.9).

The defining invariant of the alias map is that resolution never changes a value, which we record as a lemma.

Lemma G.32 (Alias resolution is value-preserving). *Suppose the pass, processing block B in frame φ , has built alias map A_i at point i , and let φ' be the target frame in the simulation at the corresponding point. Then for every value u that occurs as an operand at point i , $\llbracket \widehat{A}_i(u) \rrbracket_{\varphi'} = \llbracket u \rrbracket_\varphi$.*

Proof. By induction on the number of redundant casts processed before point i . A constant resolves to itself and is read identically in any frame. For a temporary tm : if $tm \notin \text{dom}(A_i)$ then $\widehat{A}_i(tm) = tm$, and tm is live, so the third clause of Theorem G.31 gives equality directly. If $tm \in \text{dom}(A_i)$, the entry was installed when the pass deleted a redundant cast $tm = \text{cast } v : \tau$ with $\text{type}(v) = \tau$, recording $tm \mapsto \widehat{A}_j(v)$ for the map A_j in force at that earlier point $j < i$. The semantics of an identity cast is the identity on values — $\llbracket \text{cast } v : \tau \rrbracket = \llbracket v \rrbracket$ when the cast's source and result types coincide. The only cast kinds the pass ever drops are the width- and representation-stable ones (`trunc`, `sext`, `zext` on an already-matching width, and `ptrcast`), each of which is the identity on the bit pattern; the representation-changing casts `itof` and `ftoi` can never satisfy `operand.type().equals(resultType)`, because by construction their source and result types differ (an `int`-sourced `itof` has result type `float` $\neq$ `int`, and dually for `ftoi`), so they are *vacuously* never redundant and never enter this case. So the value the source binds to tm equals $\llbracket v \rrbracket_\varphi$, which by the inner induction hypothesis equals $\llbracket \widehat{A}_j(v) \rrbracket_{\varphi'}$, i.e. $\llbracket \widehat{A}_i(tm) \rrbracket_{\varphi'}$. (Entries are invalidated exactly when a temporary they reference is redefined, so A_i never points through a stale binding.) This closes the induction. $\square$

Proof of Theorem G.30. We verify the three clauses of Theorem G.7 for $\approx$.

Clause (1). At the entry of `main` no block has been processed, every A_0 is empty, the frames are both the zero-initialized init frame, and the memories coincide; so $\text{init}(P) \approx \text{init}(P')$.

Clause (2). Let $\sigma \approx \sigma'$ with $\sigma \rightsquigarrow \tau$, and split on the source instruction at point i .

(a) *A deleted redundant cast* $tn = \text{cast } v : \tau$. The source binds $tn \mapsto \llbracket v \rrbracket_{\varphi}$ and advances to i^+ ; the target takes *zero* steps (the instruction is absent from B'). The memories and call stacks are unchanged, so they still agree. For the frame invariant: the new source binding of tn is $\llbracket v \rrbracket_{\varphi}$, and the updated alias map sets $tn \mapsto \widehat{A}_i(v)$, for which [Theorem G.32](#) gives $\llbracket \widehat{A}_i(v) \rrbracket_{\varphi'} = \llbracket v \rrbracket_{\varphi}$; every other live temporary is unaffected. Hence $\tau \approx \sigma'$, and the measure $\mathbf{m}$ drops by one (one fewer deleted cast remains), discharging the stutter condition.

(b) *Any other assignment* $tn = r$. The source evaluates r and binds tn . The target executes the corresponding rewritten assignment $tn = r'$, where r' is r with each operand replaced by its resolution. By [Theorem G.32](#) every operand of r' evaluates in φ' to the value the matching operand of r evaluates to in φ ; since the right-hand-side semantics is a function of its operand values (and of $\mu = \mu'$ for a load), the two bind the *same* word to tn . The pass also clears any alias for tn , so the new tn is live-equal on both sides and the invariant is restored. Pointer indices advance to the next corresponding point. Thus $\tau \approx \tau'$ with one target step.

(c) *A store, void call, value call, or terminator*. These are emitted with resolved operands. By [Theorem G.32](#) the resolved operands carry the same values, so a STORE writes the same word to the same address (addresses are computed from frame slots, which the pass does not touch), a CALL passes the same argument words and ENTERS the same callee with the same grown memory, and BR/JMP/RET branch on the same condition or return the same word. In each case source and target take one matching step into related successors; for a call, the pushed return contexts are related because they record corresponding continuation points and the result temporary is treated identically by both frames.

Clause (3). A terminal source state runs $\text{ret } v$ out of `main` with empty stack (or traps). The corresponding target ret returns $\llbracket \widehat{A}_i(v) \rrbracket_{\varphi'} = \llbracket v \rrbracket_{\varphi}$ by [Theorem G.32](#), so both observe the same word; a trap is matched by the same trap since the trapping `addr_index` has resolved operands of equal value. Related terminals agree on the observation.

All three clauses hold, so $\approx$ is a simulation and [Theorem G.8](#) yields $\text{obs}(P') = \text{obs}(P)$.

Well-formedness (Theorem G.11). Control-flow integrity is immediate: no block is added, removed, merged, or redirected, and no terminator changes shape (only its operands are resolved). Type agreement holds because a cast is dropped only when its operand's type equals the cast's result type, so substituting the operand for the destination never changes the static type any downstream instruction sees, and resolved operands are type-identical to those they replace. Definition dominance: every alias $tn \mapsto u$ is recorded *at* the deleted definition of tn and u resolves to a value defined no later than that point (it descends through earlier aliases or reaches a constant or a still-live temporary); since the pass's substitution is confined to a single block and the deleted temporary was itself defined and used within that block's straight-line run, every use of the substituted value is dominated by its definition. Hence $\vdash P' \text{ ok}$. $\square$

Implementation pointers

The driver is `CastSimplificationPass.simplifyBlock`; the redundancy test is `isRedundantCast (operand.type().equals(resultType))`; operand rewriting is `rewriteInstruction / rewriteRhs / rewriteTerminator`; the cycle-guarded alias chase is `resolveAlias`, and `invalidateAliases` `PointingTo` keeps the map acyclic across redefinitions.

Two notes on scope. First, the alias map is rebuilt fresh for each block (a new `HashMap` per `simplifyBlock` call), so no alias crosses a block boundary; this is what lets the per-block invariant of [Theorem G.31](#) stand without a global dominance argument, at the cost of missing cross-block identity casts. Second, the pass is narrower than the chapter’s phrase “folds cast chains” suggests: it does not algebraically cancel a `trunc` against a `sext` or an `itof` against an `ftoi`. The only chains it collapses are *identity*-cast chains, via transitive alias resolution. That keeps [Theorem G.32](#) a one-line appeal to the identity-cast semantics, with no value-changing cast ever entering the simulation — genuine narrowing of representation is left to the constant-folding and value-range passes, each proved separately.

G.7 Pass: `ControlFlowSimplificationPass`

[Section 7.3](#) describes `ControlFlowSimplificationPass` as the optimiser’s tidy-up of the control-flow graph: it folds branches whose condition is a constant Boolean, threads jumps through jump-only blocks, and merges a block into its unique successor when that successor has a unique predecessor. Each of these rewrites changes which block the program counter visits, and changes nothing else — not a temporary, not a byte of memory, not the value any `ret` ultimately produces. The whole content of this proof is that the observation function of [Theorem G.5](#) cannot see the program counter, so none of the three rewrites can move it. We discharge the obligation of [Theorem G.10](#) against the framework of [Section G.1](#).

What the pass does

Read off the source, the pass acts on one function’s block list at a time (`simplifyFunction`) and is the composition of three local rewrites: a single constant-branch FOLD, followed by the `THREAD/MERGE` pair applied to a fixed point by the outer while loop:

FOLD (`foldConstantBranches`). If a block ends in `br c,  $\ell_t, \ell_f$` and the condition `c` is the literal `#k : bool`, the terminator is replaced by `jmp  $\ell_t$` when `k  $\neq$  0` and by `jmp  $\ell_f$` when `k = 0`. Only a literal-typed-bool condition triggers the fold (`boolConst`); a branch on a temporary is left alone.

THREAD (`retargetThroughPassthrough + removePassthroughBlocks`). Call a block *passthrough* if it has *no* instructions and its terminator is an unconditional `jmp` (`passthroughLabels`). Every terminator target that names a passthrough block is rewritten to the end of the maximal chain of passthrough jumps starting there

(`resolveTarget`, which walks $\ell \rightarrow \ell' \rightarrow \dots$ and stops on a cycle or a non-passthrough block). After all targets have been redirected, every passthrough block other than the entry is deleted from the list.

MERGE (`mergeJumpToNext`). If block b ends in `jmp  $\ell$` , the textually next block is exactly ℓ , and ℓ has a unique predecessor across the whole function ($\text{predecessorCounts}(\ell) = 1$), then b and ℓ are concatenated: the merged block keeps b 's label, holds b 's instructions followed by ℓ 's instructions, and ends in ℓ 's terminator. Block ℓ is removed.

A small worked instance, the inner sieve loop of [Figure 7.1](#) abridged to its skeleton. The lowering walker emits a jump-only block L3 that does nothing but hand control to the join block L4:

```
L2 (body): ... ; jmp L3
L3:          jmp L4
L4 (join): ...
```

THREAD recognises L3 as passthrough, rewrites L2's terminator to `jmp L4`, and deletes L3. If L4 now has L2 as its only predecessor and follows it textually, MERGE folds the two into one straight-line block. The program counter takes a shorter route; the sieve computes the same primes.

The claim

Theorem G.33 (Semantic preservation of `ControlFlowSimplificationPass`). *Let P be well-formed ([Theorem G.1](#)) and let $P' = \mathcal{P}_{\text{cfs}}(P)$ be the output of `ControlFlowSimplificationPass`. Then $\vdash P' \text{ ok}$ and $\text{obs}(P') = \text{obs}(P)$.*

Because the pass is the iterated composition of FOLD, THREAD, and MERGE, and each is itself a program-to-program transformation, it suffices by [Theorem G.13](#) to prove the theorem for each of the three in isolation; the outer fixed-point loop is then a finite composition of preserving, well-formedness-preserving passes. We take the three rewrites in turn. Throughout, fix the function f being rewritten; the rewrites touch no other function, and states inside a different function step in lock-step under the identity relation.

Well-formedness

We first discharge [Theorem G.11](#) for all three rewrites, since the simulation arguments below presuppose $\vdash P' \text{ ok}$. The relevant clause of [Theorem G.1](#) is control-flow integrity: each block must end in exactly one terminator, and every branch/jump target must name a block of the same function. Type agreement and definition dominance are untouched, because no rewrite moves, deletes, or introduces an instruction *computation* — MERGE concatenates two instruction lists in their original order, and the others touch only terminators.

Lemma G.34 (FOLD, THREAD, MERGE preserve well-formedness). *If $\vdash P$ ok then each of the three rewrites yields a program in which every block ends in exactly one terminator and every terminator target names an existing block of f .*

Proof. FOLD replaces a `br` c, ℓ_t, ℓ_f by a `jmp` to one of ℓ_t, ℓ_f . Both were targets of a well-formed `br`, hence label existing blocks; the new `jmp` targets one of them, and the block still ends in exactly one terminator. No block is added or removed, so every other target is unaffected.

THREAD rewrites a target ℓ to $\text{resolve}(\ell)$, the end of the maximal passthrough chain from ℓ . Each step of `resolveTarget` follows the `jmp` terminator of an *existing* passthrough block to its target, so $\text{resolve}(\ell)$ is a label reachable by following jumps and is therefore the label of an existing block; the loop terminates because `visited` forbids revisiting a label, so a cycle of passthrough jumps stops at a block already seen rather than looping. After retargeting, no surviving terminator names a deleted block whenever the input contains no *entry-free* passthrough cycle. We treat the two cases honestly. A deleted block is passthrough, and on any chain that exits the passthrough set, `resolveTarget` maps a reference past it to a non-passthrough endpoint, which is never deleted and so remains a legal target. The dangerous case is a *pure* passthrough cycle: there `resolveTarget` stops on the first repeat and returns a label *inside* the cycle, which `removePassthroughBlocks` then deletes (it exempts only the entry, never a generic cycle member), so a surviving terminator could name a deleted block. We rule this out by a precondition the front end satisfies: every jump-only cycle FRISCCc emits is anchored at a reachable non-passthrough block — a loop header carries instructions (the loop test, the induction update), so it is not passthrough, and `resolveTarget` therefore always exits the cycle at that non-passthrough endpoint rather than looping among empty blocks. An entry-free cycle of wholly empty blocks is never produced by lowering (Theorem G.1 does not forbid it, but the lowering walker does not construct it), so the deletion never strands a target. The entry block is never deleted (`removePassthroughBlocks` skips it), so f keeps a first block. Each surviving block keeps its single terminator, now possibly with redirected targets.

MERGE concatenates b and its unique-predecessor successor ℓ into one block carrying b 's label and ℓ 's terminator; ℓ is removed. The merged block ends in exactly one terminator (ℓ 's). Every former target of b 's old `jmp` ℓ was ℓ itself, which had *unique* predecessor b ; since b is the only block that named ℓ , deleting ℓ orphans no other terminator. Any terminator that named ℓ would have been a second predecessor, contradicting $\text{predecessorCounts}(\ell) = 1$. The merged block's terminator targets are ℓ 's old targets, which name existing blocks (none of which was ℓ via b , and ℓ is the only deletion). The entry is preserved: if b is the entry, the merged block keeps b 's label and remains first. $\square$

Proof of Theorem G.33

We give a simulation (Theorem G.7) for each rewrite and invoke Theorem G.8. The common shape: a source state $\sigma = \langle (f, \ell, i), \varphi, \mu, \kappa \rangle$ is related to the target state with the *same* frame φ , memory μ , and call stack κ , and a program point that names the “same place

in the computation” under the rewrite’s relabelling. We take the anti-stuttering measure $\mathfrak{m}$ of [Theorem G.9](#) to be the number of source terminator steps not yet matched on the target side; it is bounded by the length of the longest passthrough chain (THREAD) or by 1 (MERGE), hence well-founded.

Fold. Let b be the rewritten block, with old terminator $\text{br } \#k : \text{bool}, \ell_t, \ell_f$ and new terminator $\text{jmp } \ell_t$ (the case $k \neq 0$; the $k = 0$ case is symmetric, replacing ℓ_t by ℓ_f). Define $\approx$ to be the identity on states: a source state is related to the target state with identical $\pi, \varphi, \mu, \kappa$. Initial states are identical, so clause (1) holds.

For clause (2), every program point other than b ’s terminator carries an identical instruction or terminator on both sides, and steps in lock-step by [Theorem G.4](#); the related successor is again the identity-paired state. At b ’s terminator the source steps by BR-T ([Figure G.1](#)), because the condition $\#k : \text{bool}$ evaluates to $k \neq 0$, landing at $(f, \ell_t, 0)$. The target steps by JMP on $\text{jmp } \ell_t$, also landing at $(f, \ell_t, 0)$, with φ, μ, κ unchanged on both sides. The two successors are identical, hence related; this is lock-step, so $\mathfrak{m} \equiv 0$. Clause (3): related terminals are identical and trivially observe the same word or trap. By [Theorem G.8](#), obs is preserved.

The fold relies on [Theorem G.2](#) only insofar as $\llbracket \#k : \text{bool} \rrbracket_\varphi = k$ is a constant independent of φ and μ : the static value the pass reads off the literal is exactly the value the interpreter would compute at the branch, so the arm the pass picks is the arm the interpreter would have taken. This is the only place the rewrite could go wrong, and it does not.

Thread. This is the substantive case. Let Π be the set of passthrough labels and let $\text{resolve} : \text{Label} \rightarrow \text{Label}$ be the target-rewriting function of `resolveTarget`: $\text{resolve}(\ell)$ follows the chain $\ell \rightarrow \ell_1 \rightarrow \dots$ of passthrough `jmp` targets to its first non-passthrough endpoint, or to the entry point of a passthrough cycle. The pass (i) rewrites every terminator target ℓ to $\text{resolve}(\ell)$ and (ii) deletes every non-entry passthrough block.

The key fact about a passthrough block is operational.

Lemma G.35 (Passthrough blocks are semantically transparent). *Let $\ell \in \Pi$ be a passthrough block with terminator $\text{jmp } \ell'$, and let $\sigma = \langle (f, \ell, 0), \varphi, \mu, \kappa \rangle$ be any state whose program point enters ℓ . Then $\sigma \rightsquigarrow \langle (f, \ell', 0), \varphi, \mu, \kappa \rangle$ in exactly one step, and that step leaves φ, μ , and κ unchanged.*

Proof. A passthrough block has *no* instructions, so program point 0 in ℓ is already the terminator position $\text{jmp } \ell'$. The single applicable rule is JMP ([Figure G.1](#)), which sets the pointer to $(f, \ell', 0)$ and copies φ, μ, κ verbatim. No other rule applies, by determinism ([Theorem G.4](#)). $\square$

Iterating [Theorem G.35](#) along a passthrough chain, a source run that enters ℓ traverses $\ell \rightarrow \ell_1 \rightarrow \dots \rightarrow \text{resolve}(\ell)$ in $d(\ell)$ steps, where $d(\ell) \geq 1$ is the chain length, touching only

the program counter; if the chain is a cycle, the source run diverges, spinning through the cycle forever with fixed φ, μ, κ .

Define $\approx$ as follows. A source state $\sigma = \langle (f, \ell, i), \varphi, \mu, \kappa \rangle$ is related to the target state $\sigma' = \langle (f, \hat{\ell}, \hat{i}), \varphi, \mu, \kappa \rangle$ with identical φ, μ, κ when either

1. $\ell \notin \Pi$ (the source is in a surviving block): then $\hat{\ell} = \ell$ and $\hat{i} = i$ — the same program point — *and* this surviving block is present in P' with the same instructions and a terminator whose targets are the source terminator's targets after resolve; or
2. $\ell \in \Pi$ (the source is mid-chain in a passthrough block, necessarily at $i = 0$): then $\hat{\ell} = \text{resolve}(\ell)$ and $\hat{i} = 0$ — the target has already arrived at the chain's endpoint.

Initial states: the entry block of f is never deleted, and if the entry is itself passthrough the target's entry is still that same block (its terminator merely retargeted), so $\text{init}(P) \approx \text{init}(P')$ by clause (1) of the relation (entry is a surviving block) — clause (1) of [Theorem G.7](#) holds.

Clause (2), by cases on the source step from $\sigma \approx \sigma'$.

Source in a surviving block, non-terminator step. The instruction at (f, ℓ, i) is identical on both sides (THREAD never rewrites instructions), so both step by the same rule of [Figure G.1](#) to identical φ', μ', κ' and advance to $(f, \ell, i+1)$. The successors are related by clause (1) of the relation. Lock-step; $\mathbf{m}$ unchanged.

Source in a surviving block, terminator step. The source terminator targets some ℓ_T ; the target terminator at the same block targets $\text{resolve}(\ell_T)$. If $\ell_T \notin \Pi$ then $\text{resolve}(\ell_T) = \ell_T$ and both sides jump (or branch, on the matching arm by the identical condition value) to $(f, \ell_T, 0)$; the successors are identical, related by clause (1). If $\ell_T \in \Pi$, the source steps to $(f, \ell_T, 0)$ — a passthrough block — while the target steps directly to $(f, \text{resolve}(\ell_T), 0)$. By [Theorem G.35](#) and its iteration, the source state $(f, \ell_T, 0)$ will reduce in $d(\ell_T)$ further steps to $(f, \text{resolve}(\ell_T), 0)$ with the same φ, μ, κ ; meanwhile the target is *already* at that endpoint. So we relate the source's $(f, \ell_T, 0)$ to the target's $(f, \text{resolve}(\ell_T), 0)$ by clause (2) of the relation, and the target takes *zero* steps (a stutter). The measure $\mathbf{m} = d(\ell_T) > 0$ has been freshly established; it will strictly decrease on each of the following source-only passthrough steps.

Source in a passthrough block ($\ell \in \Pi$, $i = 0$, related to target endpoint $\text{resolve}(\ell)$). By [Theorem G.35](#) the source steps to $(f, \ell', 0)$ with unchanged φ, μ, κ , where ℓ' is the next link. Either $\ell' \in \Pi$ — still mid-chain, related to the same target endpoint $\text{resolve}(\ell') = \text{resolve}(\ell)$ by clause (2), target stutters, $\mathbf{m}$ drops by one — or $\ell' \notin \Pi$, in which case $\ell' = \text{resolve}(\ell)$ and the source has now *caught up* to the target's program point; relate by clause (1) with zero target steps and $\mathbf{m} = 0$. In every sub-case the matching target multistep is empty exactly when $\mathbf{m}$ strictly decreases, satisfying [Theorem G.9](#).

Clause (3): a terminal source state is a `ret v` with empty κ (or the bounds trap). A ret terminator lives in a block that is *not* passthrough (passthrough blocks end in `jmp`), so the source is in a surviving block, related by clause (1) to the identical target program point with identical φ ; the target is at the same `ret v` and observes $\llbracket v \rrbracket_\varphi$, the same word. A

trap likewise occurs inside a non-passthrough block (it arises from a checked `addr_index` instruction, [Theorem G.2](#)), matched identically.

By [Theorem G.8](#), $\text{obs}(P') = \text{obs}(P)$ for `THREAD`. The divergence case is covered: a source run that loops through a passthrough cycle forever is matched by a target that has collapsed the cycle to a single block looping on the retargeted `jmp` — both diverge, $\perp = \perp$.

Merge. Block b , ending `jmp` ℓ , is concatenated with its unique-predecessor textual successor ℓ . Let b hold instructions $I_0, \dots, I_{p-1}$ and ℓ hold $J_0, \dots, J_{q-1}$ ending in terminator T . The merged block $\hat{b}$ (label b 's) holds $I_0, \dots, I_{p-1}, J_0, \dots, J_{q-1}$ and ends in T .

Define $\approx$ relating identical φ, μ, κ with the program-point map

$$(f, b, i) \mapsto (f, \hat{b}, i) \quad (0 \leq i \leq p), \quad (f, \ell, j) \mapsto (f, \hat{b}, p + j) \quad (0 \leq j \leq q),$$

and the identity on all other program points. The map is well-defined because the merged block's first p slots are b 's instructions in order and its next q slots are ℓ 's in order. Initial states are related (the entry is unaffected as a place, or is $\hat{b}$ itself if b was the entry).

Clause (2): for $i < p$, instruction I_i at source (f, b, i) equals the instruction at target $(f, \hat{b}, i)$; both step identically to the related pair $(f, b, i+1) \approx (f, \hat{b}, i+1)$. At the source terminator $(f, b, p) = \text{jmp } \ell$, the source steps to $(f, \ell, 0)$, which the map sends to $(f, \hat{b}, p)$ — and the target at $(f, \hat{b}, p)$ holds J_0 , the first instruction of ℓ . The target takes *zero* steps here (the source's `jmp` has no target counterpart, since the boundary is now internal to $\hat{b}$): we relate source $(f, \ell, 0)$ to target $(f, \hat{b}, p)$ directly, a single stutter with $\mathbf{m} = 1 \rightarrow 0$. Thereafter, for $j < q$, J_j at source (f, ℓ, j) equals the instruction at target $(f, \hat{b}, p + j)$ and they step in lock-step; at the source terminator $(f, \ell, q) = T$ the target terminator at $(f, \hat{b}, p + q)$ is the same T , and both take the same outgoing edge to identical successors, related by the identity on other program points. Crucially, no *other* block can jump into the middle of $\hat{b}$, because ℓ had a unique predecessor (b); had any block jumped to ℓ , it would be a second predecessor, contradicting $\text{predecessorCounts}(\ell) = 1$. So the only way to reach J_0 is through b , and the merge introduces no spurious entry into the spliced region.

Clause (3): a `ret` or `trap` in any block is matched at the same relative point with identical φ, μ , observing the same word or trap.

By [Theorem G.8](#), `MERGE` preserves `obs`.

Conclusion. `FOLD`, `THREAD`, and `MERGE` each preserve well-formedness ([Theorem G.34](#)) and the observation. Their iterated composition, which is what `simplifyFunction` computes, is preserving by [Theorem G.13](#); applying it function-by-function across P leaves every other function's runs in lock-step. Hence $\vdash P' \text{ ok}$ and $\text{obs}(P') = \text{obs}(P)$, proving [Theorem G.33](#). $\square$

Implementation pointers

The pass is `ControlFlowSimplificationPass` in `compiler-opt`. `FOLD` is `foldConstantBranches` (gated by `boolConst`, which fires only for an `IntConst` of primitive type `B00L`); `THREAD` is `retargetThroughPassthrough` (target rewriting via `resolveTarget`) followed by `removePassthroughBlocks` (deletion, with the entry block explicitly exempted); `MERGE` is `mergeJumpToNext` (gated by `predecessorCounts`). Three

remarks on where the code is more conservative than the theorem permits, none of which threatens soundness:

- `THREAD`’s `resolveTarget` guards against passthrough *cycles* with a `visited` set and stops on the first repeat, returning a label still inside the cycle. On an *entry-free* cycle of wholly empty blocks that label is then deleted, so the rewrite would strand a target; the proof of [Theorem G.34](#) rules this out by the front-end precondition that every jump-only cycle `FRISCCc` emits is anchored at a non-passthrough loop header (which carries the loop test and induction update), so `resolveTarget` always exits the cycle at that surviving endpoint. Where a genuine passthrough cycle reachable by the loop header does occur, the source program diverges through it, so the simulation’s divergence case applies and no observation is lost. The lowering walker never constructs an entry-free empty cycle, so this is a defensive guard rather than a hot path.
- `MERGE` requires the successor to be the *textually next* block (`jmp.label().equals(next.label())`), not merely a unique-predecessor successor anywhere in the list. This misses legal merges where the successor sits elsewhere in the block list; it is a deliberate simplification that keeps the rewrite a local splice and never merges across a relabelling. Soundness is unaffected — the proof only ever uses the textual-adjacency case — and the missed opportunities are recovered after `THREAD` reorders nothing but is followed by the outer fixed-point loop.
- `FOLD` folds only a literal `bool` condition; a branch on a temporary that provably holds a constant is left for `GlobalValuePropagationPass` or `ValueRangeSimplificationPass` to constant-fold first. The division of labour matches [Section 7.3](#)’s “recognise then remove” rhythm and keeps this pass’s correctness argument confined to the program counter.

The framework’s [Theorem G.12](#) gives termination preservation for free, and the cooperation with `UnreachableBlockEliminationPass` — which physically removes the blocks `THREAD` orphans — is proved in [Section G.8](#). The textbook framing of jump threading as a control-flow-graph transformation that preserves the trace projected to observable events is Muchnick’s [\[62, §18.2\]](#) and Cooper and Torczon’s [\[18, §10.2\]](#); the simulation-relation discipline we use to make it precise is `CompCert`’s [\[56\]](#).

G.8 Pass: UnreachableBlockEliminationPass

Section 7.3 describes `UnreachableBlockEliminationPass` as the obvious follow-up to control-flow simplification: it computes, by a graph traversal from the entry block, the set of blocks reachable along terminator edges, and deletes every block not in that set. The correctness story is the shortest in the appendix, and the chapter states it in one sentence — “no execution of the program ever runs an unreachable block’s instructions.” This proof makes that precise against the framework of Section G.1: a block that is unreachable in the *static* control-flow graph is never the current block of any *runtime* state the program actually visits, so removing it deletes only states the simulation never relates to anything.

What the pass does

Per function (`eliminate`), let ℓ_0 be the label of the first block — the entry. Define the static successor relation on labels by reading each block’s terminator (`successors`):

$$\text{succ}(\ell) = \begin{cases} \{\ell'\} & \text{terminator of } \ell \text{ is } \text{jmp } \ell', \\ \{\ell_t, \ell_f\} & \text{terminator is } \text{br } c, \ell_t, \ell_f, \\ \emptyset & \text{terminator is } \text{ret } [v]. \end{cases}$$

Let $\mathcal{R}$ be the set of labels reachable from ℓ_0 under the reflexive-transitive closure of `succ` — exactly the worklist closure computed by `computeReachable`. The pass keeps the blocks whose label is in $\mathcal{R}$, in their original order, and drops the rest. If $\mathcal{R}$ already covers every block the function is returned unchanged.

This is a purely *syntactic* notion of reachability, as Section 7.3 stresses: the traversal follows *both* arms of every `br` regardless of whether the condition could ever select that arm at run time. A block guarded by an always-false comparison is *not* deleted here — that is the downstream job of `ValueRangeSimplificationPass`. Consequently the deleted blocks are exactly those with no syntactic path from the entry at all, and these are the blocks that no run can ever enter. The pass typically removes the join blocks that `ControlFlowSimplificationPass` orphans when it threads jumps, and the else-arm a constant-true fold leaves dangling.

The claim

Theorem G.36 (Semantic preservation of `UnreachableBlockEliminationPass`). *Let P be well-formed (Theorem G.1) and let $P' = \mathcal{P}_{\text{ube}}(P)$ be the output of `UnreachableBlockEliminationPass`. Then $\vdash P'$ ok and $\text{obs}(P') = \text{obs}(P)$.*

Well-formedness

Lemma G.37 (Elimination preserves well-formedness). *If $\vdash P$ ok then $\vdash P'$ ok.*

Proof. We check the relevant clauses of [Theorem G.1](#) for each rewritten function f ; the pass deletes whole blocks but rewrites no instruction, terminator, slot, or type, so type agreement is inherited verbatim and we need only control-flow integrity and a non-empty block list with a legal entry.

Entry survives and has no incoming edge. The entry label ℓ_0 is the root of the reachability traversal, so $\ell_0 \in \mathcal{R}$ and the entry block is kept; it remains the first block because deletion preserves order. Well-formedness of P already guaranteed the entry has no predecessor in f ([Theorem G.1](#)); deleting blocks only removes terminators, so it cannot *create* an edge into ℓ_0 .

Every surviving terminator targets a surviving block. Suppose a kept block $\ell \in \mathcal{R}$ has a terminator naming target ℓ' . Then $\ell' \in \text{succ}(\ell)$, and since ℓ is reachable from ℓ_0 , so is ℓ' by one more step of succ ; hence $\ell' \in \mathcal{R}$ and ℓ' is kept. So no surviving terminator can point at a deleted block. Each kept block keeps its single original terminator unchanged, so the “exactly one terminator” condition is preserved.

Definition dominance. A use of a temporary in a kept block was dominated, on every entry-to-use path in P , by a definition of it. Deleting unreachable blocks removes no block on any such path — every block on a path from ℓ_0 is, by construction, in $\mathcal{R}$ — so the dominating definitions all survive. The dominance invariant of [Theorem G.1](#) is therefore preserved. $\square$

Proof of Theorem G.36

The heart of the matter is that the runtime never visits a deleted block. We make “visit” precise and prove an invariant on reachable runtime states.

Definition G.38 (Current block of a state). For a state $\sigma = \langle (f, \ell, i), \varphi, \mu, \kappa \rangle$, the *current label* is ℓ and, for the topmost-and-deeper frames, the labels recorded in the return contexts of κ are the *pending labels*: the block each suspended caller will resume in. The *live labels* of σ are the current label together with all pending labels, each paired with the function it belongs to.

Lemma G.39 (Reachable runtime states live only in static-reachable blocks). *Let P be well-formed and let σ be any state with $\text{init}(P) \rightsquigarrow^* \sigma$. Then every live label of σ lies in the static-reachable set $\mathcal{R}_g$ of its own function g (the set the pass computes for g).*

Proof. By induction on the length of the run $\text{init}(P) \rightsquigarrow^* \sigma$.

Base. $\text{init}(P)$ has current label ℓ_0^{main} , the entry of main , and empty call stack, so its only live label is the root $\ell_0^{\text{main}} \in \mathcal{R}_{\text{main}}$.

Step. Assume the invariant for σ and let $\sigma \rightsquigarrow \tau$. We case on the rule of Figure G.1 that fires, tracking how the current and pending labels change. Let g be σ 's current function and ℓ its current label; by hypothesis $\ell \in \mathcal{R}_g$.

- **ASSIGN-OP / STORE** advance the pointer within the same block (ℓ unchanged) and do not touch κ ; the live labels are unchanged, so the invariant carries over.
- **JMP** on `jmp  $\ell'$` makes ℓ' the new current label. Since $\ell' \in \text{succ}(\ell)$ and $\ell \in \mathcal{R}_g$, one more succ step gives $\ell' \in \mathcal{R}_g$. Pending labels and their functions are untouched.
- **BR-T / BR-F** make one of ℓ_t, ℓ_f the new current label. Both are in $\text{succ}(\ell)$, so whichever the condition selects is in $\mathcal{R}_g$ by the same one-step argument.
- **CALL** pushes a return context recording the caller's *continuation point*, whose label is the caller's current label ℓ — already in $\mathcal{R}_g$ by hypothesis — and transfers control to the callee's *entry* block ℓ_0^h , the root of $\mathcal{R}_h$. So the new current label $\ell_0^h \in \mathcal{R}_h$, and the freshly pushed pending label $\ell \in \mathcal{R}_g$; the older pending labels are unchanged.
- **RET-CALLER** pops the top return context and resumes the caller at its recorded continuation label, which was a pending label of σ and hence (by hypothesis) in $\mathcal{R}$ of the caller's function. The callee's current label disappears; the remaining live labels were already covered.

In every case the live labels of τ lie in the appropriate static-reachable sets, closing the induction. $\square$

Theorem G.39 is the whole soundness story: the running program only ever *is in*, and only ever *remembers*, blocks that the pass keeps. A deleted block is, by definition of $\mathcal{R}$, not reachable from the entry under succ, so it is never a live label of any reachable state — the deletion removes program text that no run touches.

We now exhibit the simulation. Take $\approx$ to be the *identity on states* restricted to states all of whose live labels survive in P' :

$$\sigma \approx \sigma' \iff \sigma = \sigma' \text{ and every live label of } \sigma \text{ is kept in } P'.$$

Since the kept blocks of P' carry exactly the same instructions and terminators as in P (the pass copies, never edits, surviving blocks), “ $\sigma = \sigma'$ ” is meaningful: a state over the kept blocks is literally a state of both programs. Take the anti-stuttering measure $\mathfrak{m} \equiv 0$ (this is a true lock-step simulation, so Theorem G.9 is discharged vacuously).

Clause (1). $\text{init}(P) = \text{init}(P')$: both enter `main`'s entry block, which is kept (Theorem G.37), with the same fresh μ , empty ρ , and empty κ . The single live label ℓ_0^{main} is kept, so the two initial states are related.

Clause (2). Suppose $\sigma \approx \sigma'$, i.e. $\sigma = \sigma'$ with all live labels kept, and $\sigma \rightsquigarrow \tau$. Because $\text{init}(P) \rightsquigarrow^* \sigma$ along the run we are tracking, Theorem G.39 applies to σ and to its successor τ : every live label of τ is static-reachable, hence kept in P' . The instruction or terminator

at σ 's current point is identical in P' (the block survives unedited), so the same rule of Figure G.1 fires from the identical state $\sigma' = \sigma$ in P' , producing the identical successor τ . Thus $\sigma' \rightsquigarrow \tau$ with $\tau \approx \tau$ — a single matching target step. Lock-step; the measure is constant.

Clause (3). A terminal source state is a `ret` v with empty κ , or the bounds trap; in either case it is the identical state in P' (its block survives), so the target is already terminal and observes the same word $\llbracket v \rrbracket_\varphi$ or the same trap `-6`.

By Theorem G.8, $\text{obs}(P') = \text{obs}(P)$. The divergence case is automatic: an infinite source run visits only kept blocks (Theorem G.39), and the identical sequence of steps is an infinite run of P' , so both diverge. With Theorem G.37 this proves Theorem G.36. $\square$

Remark G.40 (Why deletion changes the text but not the trace). The proof exposes exactly the gap Section 7.3 alludes to. The static text of P and P' differ — P' has fewer blocks — but their operational *traces* from `init` are step-for-step identical, because the deleted blocks contribute no states to either trace. The observation function, which projects a trace onto its terminal return value or trap (Theorem G.5), therefore cannot distinguish them. This is the cleanest possible instance of the appendix's recurring theme: a transformation is safe precisely when its effect lies entirely inside the part of the state that `obs` discards.

Implementation pointers

The pass is `UnreachableBlockEliminationPass` in `compiler-opt`. The reachable set $\mathcal{R}$ is computed by `computeReachable`, a breadth-first worklist over labels seeded with the entry label (`blocks.getFirst().label()`) and expanded through successors; `eliminate` then filters the block list to $\mathcal{R}$, preserving order, and returns the function unchanged when $|\mathcal{R}|$ already equals the block count. Two notes on the correspondence between code and theorem:

- `computeReachable` silently skips a successor label with no matching block (if `(block == null)` `continue`). On a well-formed input this branch is dead — Theorem G.1 guarantees every terminator target names an existing block — so it does not affect $\mathcal{R}$ as defined above. It is a defensive guard that keeps the pass total on malformed input rather than a semantic choice; the proof assumes the well-formed precondition under which it never fires.
- The pass is purely syntactic, as Section 7.3 warns and Theorem G.39's use of *both* br arms reflects: it deletes only blocks with no static path from the entry, never a block that is dynamically dead because its guarding condition is always false. Deleting the latter would require the value-range reasoning of `ValueRangeSimplificationPass` and a correspondingly stronger argument; the conservative syntactic notion used here is what makes the lock-step simulation above so short.

The framework's Theorem G.12 supplies termination preservation, and Theorem G.13 licenses running this pass after `ControlFlowSimplificationPass` (Section G.7), whose threaded

jumps orphan the very blocks this pass removes — the two-step “recognise then remove” rhythm of [Section 7.3](#), now both halves proved. The treatment of unreachable-code elimination as a reachability fixpoint over the control-flow graph follows Appel [7, §17.2], Muchnick [62, §18.1], and Cooper and Torczon [18, §10.2].

G.9 Pass: global value propagation

What the pass does

Where copy propagation ([Section G.10](#)) reasons inside a single block, global value propagation reasons across a whole function’s control-flow graph. It is a textbook *forward dataflow analysis* followed by a rewrite, and its correctness is the cleanest instance in this appendix of the abstract-interpretation discipline that [Chapter 7](#) frames the optimizer with: the analysis computes, at every program point, a finite *fact* that over-approximates the set of runtime states that can reach that point, and the rewrite only ever replaces a use by a value the fact certifies it must already hold. Soundness of the rewrite reduces to soundness of the analysis, which is the Cousot–Cousot fixed-point recipe [20] specialized to a small constant/copy lattice, computed by the classic worklist iteration of Kildall [44]. The implementation is `GlobalValuePropagationPass` in `compiler-opt`.

Concretely, the pass tracks the contents of *tracked slots* — the local and parameter slots of primitive type (`trackedSlots`) — as they flow through stores and loads. A fact maps each tracked slot to one of: a known constant (`ConstFact`), a known alias to another tracked slot (`CopyFact`), or no information (absence from the map). The block transfer function (`transferBlock`) walks a block, threading these slot facts and a local map of temporary facts together: a store to a tracked slot records what was written; a load from a slot known-constant yields a constant right-hand side; constant operands fold through integer arithmetic and comparisons; and any event the analysis cannot model — a call, an increment/decrement, a store through an unresolved address — *conservatively clears* the slot facts (`slotState.clear()`). At control-flow joins the incoming facts are *merged* by agreement: a slot keeps a fact only if every predecessor agrees on it (`mergePredecessors`). The analysis iterates to a fixed point, and a second pass over the blocks rewrites loads-of-constant to constants, folds operations on now-constant operands, and substitutes constant temporaries into operands (`rewriteRhs`, `rewriteValue`). For example, a function that stores #7 into a local x on both arms of a branch and later loads x has the load rewritten to #7, because the merge of the two arms still certifies $x = \#7$.

The claim

Theorem G.41 (Global value propagation preserves semantics). *Let $\mathcal{P}_{\text{gvp}}$ be the global-value-propagation pass. For every well-formed program P ([Theorem G.1](#)), $\text{obs}(\mathcal{P}_{\text{gvp}}(P)) = \text{obs}(P)$.*

The pass changes *no* control flow and adds, deletes, or reorders *no* instructions: it only rewrites right-hand sides and operands in place. Consequently P and $P' = \mathcal{P}_{\text{gvp}}(P)$ run in exact lock-step — one source step to one target step — and the entire burden is to show that each rewritten right-hand side evaluates to the same value as the original in every reachable state. This is the soundness of the analysis.

The abstraction and its soundness

We make precise what a fact *means*. Recall from [Section G.1](#) that the address of a tracked slot s in a frame $\varphi = (\rho, \beta)$ is fixed by β ; write $\text{addr}_\varphi(s)$ for it, and $\text{load}_\tau(\mu, \cdot)$ for the typed read of [Figure G.1](#).

Definition G.42 (Concretization of a fact). A *fact* Φ is a finite partial map from tracked slot names to elements $\{\text{ConstFact}(k : \tau)\} \cup \{\text{CopyFact}(s')\}$. A state $\sigma = \langle (f, \ell, i), (\rho, \beta), \mu, \kappa \rangle$ is *consistent with* Φ , written $\sigma \models \Phi$, when for every slot $s \in \text{dom } \Phi$ of type τ :

$$\begin{aligned} \Phi(s) = \text{ConstFact}(k : \tau) &\implies \text{load}_\tau(\mu, \text{addr}_\varphi(s)) = k, \\ \Phi(s) = \text{CopyFact}(s') &\implies \text{load}_\tau(\mu, \text{addr}_\varphi(s)) = \text{load}_\tau(\mu, \text{addr}_\varphi(s')). \end{aligned}$$

$\gamma(\Phi) := \{\sigma : \sigma \models \Phi\}$ is the *concretization* of Φ in the sense of abstract interpretation [\[20\]](#): the set of runtime states the fact claims to describe.

The ordering on facts is reverse map-inclusion refined by element equality: $\Phi_1 \sqsubseteq \Phi_2$ (i.e. Φ_1 is *stronger*) when $\text{dom } \Phi_2 \subseteq \text{dom } \Phi_1$ and the two agree on $\text{dom } \Phi_2$. The merge `mergePredecessors` computes the meet: it keeps a slot binding only when all predecessors carry the identical binding. Monotonicity of γ in this order ($\Phi_1 \sqsubseteq \Phi_2 \Rightarrow \gamma(\Phi_1) \subseteq \gamma(\Phi_2)$) is immediate from [Theorem G.42](#): dropping a slot from the domain only enlarges the described set.

Lemma G.43 (Transfer soundness). Let Φ be the analysis fact holding at the entry of a block B , and let Φ' be the fact `transferBlock` produces at B 's exit. If $\sigma \models \Phi$ and σ runs through B to a state τ at B 's exit (under the original program's steps), then $\tau \models \Phi'$. The same holds for every intermediate point of B against the intermediate fact the transfer carries there.

Proof. By induction on the instructions of B , tracking the transfer's three maps — the slot facts `slotState`, the per-temporary facts `tempFacts`, and the per-temporary address facts `tempAddresses` — against the running state. The induction hypothesis at each point is the conjunction of $\sigma' \models \text{slotState}$ ([Theorem G.42](#)); *temporary soundness*: for every $tn \in \text{dom}(\text{tempFacts})$, $\llbracket tn \rrbracket_{\varphi'}$ equals the constant or the slot-load that the temporary fact names; and *address soundness*: for every $tn \in \text{dom}(\text{tempAddresses})$, $\llbracket tn \rrbracket_{\varphi'} = \text{addr}_{\varphi'}(\text{tempAddresses}[tn])$ — the temporary holds the address of the named tracked slot. The `tempAddresses` map is populated only when a right-hand side is an `AddrOfSymbol` of a tracked slot, where address soundness is immediate from the `ADDR-OF` rule, and it is the

sole mechanism by which a load or store address temporary is resolved back to a tracked slot (`resolveAddress`). We check each instruction form against `transferBlock`.

Store to a tracked slot s (`IrStoreInstr`). The transfer resolves the store’s address temporary through `tempAddresses` (`resolveAddress`); by address soundness that temporary holds $\text{addr}_\varphi(s)$, so the resolved slot s is exactly the slot the STORE rule writes. The transfer then sets `slotState[s]` to the fact of the stored value (a `ConstFact` if the value is a known constant, a `CopyFact` if it is a known slot alias, normalized to the slot’s type by `normalizeForSlot`), and the STORE rule writes exactly that value to $\text{addr}_\varphi(s)$. So the new `slotState[s]` is consistent by construction; bindings of *other* slots are unaffected because distinct tracked slots occupy distinct addresses (the slot table is injective on tracked names per [Theorem G.1](#)), so the store cannot disturb another slot’s load. A store whose address temporary `tempAddresses` does not resolve to a tracked slot clears `slotState` entirely, which is conservatively sound: it makes no claim about any slot, so consistency holds vacuously regardless of where the store actually landed.

Load from a tracked slot (`IrRhs.Load`). The transfer resolves the load’s address temporary through `tempAddresses`; by address soundness it names a tracked slot s whose address the LOAD rule reads. If s ’s fact is `ConstFact( $k$ )`, the load reads k by hypothesis $\sigma' \models \text{slotState}$, so recording `tempFacts[dest] = ConstFact( $k$ )` preserves temporary soundness; the `CopyFact` case is analogous through the alias’s load value. A load whose address does not resolve to a tracked slot records no temporary fact, trivially preserving soundness.

Constant, arithmetic, comparison right-hand sides. A constant right-hand side records the literal; an integer binop or comparison both of whose operands carry constant facts is folded (`foldIntBinOp`, `compare`) to the integer the ASSIGN-OP rule would itself compute, by [Theorem G.2](#) (and the transfer declines to fold `div` or `mod` by zero — it returns no fact — so it never records a value the semantics would not produce); and an integer unary negation of a constant operand folds to its arithmetic negation (the `NEG` case of `evaluateRhsFact`), matching the ASSIGN-OP rule for unary minus. Algebraic shortcuts ($x + 0$, $0 + x$, $x - 0$) record the operand’s own fact, which equals the result. Each recorded temporary fact thus matches the value ASSIGN-OP binds, preserving temporary soundness.

Conservative clears. A call, an increment/decrement, or any right-hand side flagged by `hasUnknownMemoryWrite` clears `slotState`, and a void or value call clears it as well. Clearing empties `dom(slotState)`, so $\models$ holds vacuously after; this is the conservative over-approximation that keeps the analysis sound across memory effects it does not model.

In every case the post-instruction fact is consistent with the post-instruction state, completing the induction; at B ’s exit the carried fact is Φ' , and $\tau \models \Phi'$. $\square$

Lemma G.44 (Analysis soundness at every reachable point). *After the worklist iteration reaches its fixed point, for every block B the entry fact `inStates[B]` is consistent with every state that reaches B ’s entry along any run of P from `init(P)`.*

Proof. The fixed point satisfies the standard dataflow equations: the entry fact of the first block is $\top$ (the empty map, consistent with every state), and the entry fact of every other

block is the merge (`mergePredecessors`) of the exit facts of its predecessors, each exit fact being the transfer of the corresponding entry fact ([Theorem G.43](#)). The entry block’s fact is fixed to $\top$ regardless of any predecessor edges — the pass overrides its incoming merge unconditionally — which is sound because $\text{init}(P)$ enters with no established slot facts, so $\top$ over-approximates the single initial state even if a back-edge targets the entry block. We show by induction on the length of a run that whenever a run reaches a program point p in state σ , $\sigma \models \text{fact}(p)$, where $\text{fact}(p)$ is the analysis fact carried at p .

The base case is the entry of the entry block: $\text{init}(P)$ enters with fact $\top$, and $\sigma \models \top$ vacuously. For the step, suppose the claim holds at p and the run moves to p^+ . If p and p^+ are in the same block, [Theorem G.43](#) (its “intermediate point” clause) gives $\sigma^+ \models \text{fact}(p^+)$. If p is a terminator transferring control to a block B' , then the state at B' ’s entry is consistent with the transfer’s exit fact for p ’s block ([Theorem G.43](#) at the block exit); since the merge that defines $\text{inStates}[B']$ is the meet *over all* predecessors, it is $\sqsubseteq$ (weaker than) each predecessor’s contribution, and by monotonicity of γ the reaching state is consistent with the merged $\text{inStates}[B']$. (Calls and returns preserve $\models$ for the caller’s frame because the transfer clears slot facts across a call, so the post-return fact makes no claim that the callee’s memory writes could falsify; the callee runs under its own analysis on its own frame.) Hence consistency holds at p^+ . $\square$

The simulation relation and the rewrite

Definition G.45 (GVP simulation). Relate σ of P to σ' of P' , written $\sigma \approx \sigma'$, when they are *identical* states: same pointer, same frame, same memory, same call stack. The anti-stuttering measure is $\mathfrak{m} \equiv 0$ (true lock-step; [Theorem G.9](#) is vacuous).

Identity is a candidate simulation only because the rewrite never changes which value any instruction computes — it merely changes *how* the value is written in the IR text. The next lemma is exactly that fact.

Lemma G.46 (Rewrite value-preservation). *Let an instruction at point p be rewritten by `rewriteRhs` / `rewriteValue` using the fixed-point fact $\Phi = \text{fact}(p)$, and let σ be any state reaching p . Then the rewritten right-hand side (resp. operand, branch condition, return value) evaluates in σ to the same value as the original.*

Proof. By [Theorem G.44](#), $\sigma \models \Phi$. The rewrite pass re-runs `transferBlock` from the fixed-point entry fact $\text{inStates}[B]$ ([Theorem G.44](#) gives $\sigma \models \text{inStates}[B]$); since the rewrite does not alter which facts the transfer derives on the prefix — it only substitutes constants into already-derived right-hand sides — its local `tempFacts` coincide with the analysis pass’s on the prefix and inherit temporary soundness for σ by the same induction as [Theorem G.43](#). The rewrite performs only these replacements, each value-preserving under that soundness:

Load of a known-constant slot $\mapsto$ that constant: by $\sigma \models \Phi$, $\text{load}_\tau(\mu, \text{addr}_\varphi(s)) = k$, and the rewrite checks the constant’s type equals the load type (`constant.type().equals(`

`load.loadType()`), so the constant right-hand side evaluates to the same word the load reads.

Constant temporary into an operand (`rewriteValue`): replaces `tn` by `ConstFact(k)`'s literal only when the temporary fact is a constant and (where an expected type is supplied) the types match; by temporary soundness $\llbracket tn \rrbracket_{\varphi} = k$, so the operand reads identically.

Folded arithmetic / comparison: `evaluateRhsFact` folds only constant operands, and `foldInt BinOp/compare` use the same `int32` operations and 0/1 comparison indicators as Figure G.1, including the Theorem G.2 treatment (and declining to fold division or modulo by zero); likewise an integer unary negation of a constant operand folds to its arithmetic negation (the `NEG` case of `evaluateRhsFact`), matching the `ASSIGN-OP` rule for unary minus. In each case the folded constant equals the value the original right-hand side would compute.

No other right-hand side, store, terminator label, or block is touched. Hence every rewritten form evaluates to the original's value in σ . $\square$

Discharging the obligations

Proof of Theorem G.41. Well-formedness. The pass keeps the block list, all labels, and all terminators structurally identical (it only swaps a `br` or `ret` *operand*, never a label), so control-flow integrity holds; it deletes no definition, so definition dominance holds; and every substituted constant is coerced to the consumer's type (`coerceConstant`, the type guards in `rewriteValue` and `rewriteRhs`), so type agreement holds. Thus $\vdash \mathcal{P}_{\text{gvp}}(P)$ *ok* by Theorem G.11.

Simulation. Take $\approx$ to be state identity (Theorem G.45). Clause (1): `init(P)` and `init(P')` are identical, since the pass changes no slot table, no globals, and no entry pointer. Clause (2): suppose $\sigma \approx \sigma'$, i.e. $\sigma = \sigma'$, and $\sigma \rightsquigarrow \tau$. The instruction or terminator at this point is the same form on both sides; only its operands or right-hand side may differ textually. By Theorem G.46 the rewritten form evaluates in $\sigma' = \sigma$ to the same value(s) as the original, so the matching rule of Figure G.1 produces an identical successor: the same word is bound, the same address is written with the same value, the same branch direction is taken, the same callee is entered with the same argument words. Hence $\sigma' \rightsquigarrow \tau'$ with $\tau' = \tau$, so $\tau \approx \tau'$ in true lock-step. Clause (3): a terminal `ret v` with empty κ matches the identical state on the target with a return value that evaluates equally (Theorem G.46), and a bounds trap matches a bounds trap because the index operand evaluates equally.

All clauses hold, so by Theorem G.8 $\text{obs}(\mathcal{P}_{\text{gvp}}(P)) = \text{obs}(P)$; termination and trapping behaviour follow from Theorem G.12. $\square$

Implementation pointers

The pass's `rewriteFunction` runs the worklist to a fixed point (`inStates/outStates` with the stable flag) and then rewrites in a second pass with `rewrite = true`. The lattice elements

are `ConstFact` and `CopyFact`; `mergePredecessors` is the `meet`; `transferBlock` is the transfer function; `coerceConstant` enforces the type guards that make [Theorem G.46](#) hold. Three honest notes on the gap between code and theorem. First, the analysis is deliberately *conservative*, not complete: it clears all slot facts on any call, increment/decrement, or unresolved store, so it forgets facts a more precise alias analysis could keep — this only ever weakens Φ , which keeps $\gamma(\Phi)$ a sound over-approximation and never causes an unsound rewrite. Second, division and modulo by zero are never folded into a fact (`foldIntBinOp` returns `null`), so the pass cannot manufacture the framework’s total-arithmetic value 0 where the original program would have computed it dynamically; it simply leaves the operation in place, which is trivially value-preserving. Third, only primitive local and parameter slots are tracked (`trackedSlots`); aggregates, globals, and spills are outside the abstraction’s domain and contribute no facts, so the relation reduces to identity on every instruction the pass leaves untouched.

G.10 Pass: copy propagation

What the pass does

Copy propagation is the smallest of the temporary-level rewrites, and a good place to see the framework of [Section G.1](#) do its work on a pass that changes almost nothing. It operates one block at a time. As it scans a block top to bottom it maintains an *alias map* A from temporary indices to values: whenever it meets an assignment $tb = r$ whose right-hand side r is an *identity copy* of some value a — and whose destination tb is used exactly once in the rest of the block — it records $A[tb] = a$, drops the assignment, and rewrites every later mention of tb to a instead. Other instructions are left in place but have their operands rewritten through A . The implementation is `CopyPropagationPass` in `compiler-opt`; the alias chasing is `resolveAlias` and the identity test is `identitySource`.

What counts as an identity copy is deliberately narrow, and it is fixed by `identitySource`. On an `int32` binary operation it recognizes $x + \#0$, $\#0 + x$, $x - \#0$ (additive identity), $x \cdot \#1$, $x / \#1$ (multiplicative identity), and $x \ll \#0$, $x \gg \#0$ (shift by zero); on a cast it recognizes a pointer cast `ptrcast  $x : \tau$` whose operand already has type τ . In every case the right-hand side denotes exactly the value of its surviving operand x . For instance, the block

```
t3 = t2 + #0 : int32
t5 = t3 * t4 : int32
ret t5
```

becomes

```
t5 = t2 * t4 : int32
ret t5
```

once $t3$ ’s single use in $t5 = t3 \cdot t4$ has been rewritten to $t2$ and the now-redundant copy deleted.

The claim

Theorem G.47 (Copy propagation preserves semantics). *Let $\mathcal{P}_{\text{copy}}$ be the copy-propagation pass. For every well-formed program P (Theorem G.1), $\text{obs}(\mathcal{P}_{\text{copy}}(P)) = \text{obs}(P)$.*

We discharge Theorem G.47 through Theorem G.10: we check that the pass preserves well-formedness, exhibit a simulation relation, and verify its three clauses. The work concentrates in a single invariant about what the alias map means at runtime.

Well-formedness

Lemma G.48 (Copy propagation preserves $\vdash \cdot \text{ok}$). *If $\vdash P \text{ok}$ then $\vdash \mathcal{P}_{\text{copy}}(P) \text{ok}$.*

Proof. We check the three sub-obligations of Theorem G.11. The pass touches no terminator structure — it rewrites the operands of a `br` or `ret` but never changes a label, and never adds, deletes, or reorders blocks — so *control-flow integrity* is immediate. For *definition dominance*, the only deleted instruction is a copy `tb = r` whose destination `tb` is used exactly once *within the same block* (`useCounts.getOrDefault(...) == 1`); that single use is rewritten to the copy’s source value `a`, and every operand `a` substituted by `resolveAlias` is the source of an identity copy that textually precedes the use, hence is itself dominated by its own definition (a constant operand has no definition obligation). No use of `tb` survives the rewrite, so deleting its definition leaves every remaining use dominated. *Type agreement* holds because each recognized identity copy denotes a value of the destination’s type — additive and multiplicative identities on `int32` stay `int32`, and the recognized `ptrcast` $x : \tau$ has $x : \tau$ already (`identitySource` checks `operand().type().equals(resultType())`) — so substituting `a` for `tb` replaces a value of a type by a value of the same type, and every rewritten instruction still type-checks. $\square$

The simulation relation

The single use restriction is what lets the pass be *forward*: an aliased temporary is read at most once after it is bound, so substituting the source value at that one use is observationally equivalent to binding the temporary and reading it. We make “*A* tells the truth” precise.

Definition G.49 (Copy simulation). Fix a block under rewrite, let A be the alias map at a given program point, and let $P' = \mathcal{P}_{\text{copy}}(P)$. Relate $\sigma = \langle \pi, (\rho, \beta), \mu, \kappa \rangle$ of P to $\sigma' = \langle \pi', (\rho', \beta), \mu', \kappa' \rangle$ of P' , written $\sigma \approx \sigma'$, when:

1. the memories agree, $\mu = \mu'$, and the call stacks agree up to the same relation applied frame-by-frame;
2. the two pointers are at *corresponding* program points — σ' is at the surviving image of σ ’s instruction, skipping deleted copies; and

3. for every temporary tn that is *live* at the current point (read somewhere in the remainder of the block), $\llbracket tn \rrbracket_{(\rho, \beta)} = \llbracket A^*(tn) \rrbracket_{(\rho', \beta)}$, where A^* is the fixed point of alias chasing (resolveAlias): $A^*(tn) = tn$ if $tn \notin \text{dom } A$, else $A^*(A[tn])$.

The anti-stuttering measure $\mathbf{m}(\sigma, \sigma')$ is the number of deleted copies between σ 's point and the next surviving instruction (a bounded count within one block), so [Theorem G.9](#) is satisfied.

Clause (3) is the heart of the matter: a live temporary in the original has the same runtime value as its alias representative in the rewritten program. Because temporaries in this IR are block-local — defined and consumed within a single block, never carried across a block boundary ([Section 6.5](#)) — the live set at any point is exactly the in-block remainder, and the single-use deletion guard, which counts in-block instruction and terminator uses (`useCounts.getDefault(...) == 1`), is therefore complete: a temporary the guard sees used once is used exactly once, period. The relation holds at block entry because A is empty there, so A^* is the identity and the two temporary environments are literally equal.

Lemma G.50 (Alias soundness). *Suppose $\sigma \approx \sigma'$ at a program point p , and let p host an identity-copy assignment $tb = r$ with recognized source a (so $\llbracket r \rrbracket_{(\rho, \beta), \mu} = \llbracket a \rrbracket_{(\rho, \beta)}$) and tb used once after p . The pass deletes this instruction on the P' side and extends A to $A[tb] = A^*(a)$. Then after σ takes its single ASSIGN-OP step to τ , the unchanged σ' stands in the relation: $\tau \approx \sigma'$ under the extended A .*

Proof. The source step binds $\rho(tb) \mapsto w$ where $w = \llbracket r \rrbracket_{(\rho, \beta), \mu}$; memory, call stack, and all other temporaries are unchanged, and the P' side does not move (the deleted copy is a pure stutter, and $\mathbf{m}$ drops by one). We must re-establish clause (3) for $\tau \approx \sigma'$. For a live temporary other than tb , its value and its alias representative are both unchanged, so the equality survives. For tb itself: by the identity-copy recognition (`identitySource`), r denotes exactly the value of its surviving operand a , so $w = \llbracket a \rrbracket_{(\rho, \beta)}$; by the relation at p applied to the (live) operand a , $\llbracket a \rrbracket_{(\rho, \beta)} = \llbracket A^*(a) \rrbracket_{(\rho', \beta)}$. The extended map sets $A^*(tb) = A^*(a)$, so $\llbracket tb \rrbracket_{\tau} = w = \llbracket A^*(tb) \rrbracket_{(\rho', \beta)}$, which is exactly clause (3) for tb . (The single-use condition guarantees tb is never rebound before its one use, so A stays consistent until that use is consumed.) $\square$

Discharging the simulation clauses

Proof of Theorem G.47. By [Theorem G.48](#) both programs are well-formed, so [Theorem G.8](#) applies once we exhibit the simulation $\approx$ of [Theorem G.49](#) and check its clauses.

Initial states (clause 1). A is empty at every block entry, so A^* is the identity; $\text{init}(P)$ and $\text{init}(P')$ have the same empty temporary environment, the same fresh μ , and the same entry pointer, hence $\text{init}(P) \approx \text{init}(P')$.

Steps (clause 2). Fix $\sigma \approx \sigma'$ with $\sigma \rightsquigarrow \tau$, and case on the instruction at σ 's point.

Deleted copy. If the point hosts a recognized identity copy that the pass deleted, [Theorem G.50](#) gives $\tau \approx \sigma'$ with the P' side stuttering (zero target steps), and $\mathbf{m}$ strictly decreases.

Any surviving instruction or terminator. Otherwise the instruction survives on the P' side with its operands rewritten through A^* . By clause (3) of the relation, each operand value v read on the source side equals $\llbracket A^*(v) \rrbracket_{(\rho', \beta)}$ on the target side — this covers operands of a binop, comparison, unary, cast, load address, store value and address, call arguments, branch condition, and return value, since `IrValueRewriter` substitutes *every* value operand uniformly. Each rule of [Figure G.1](#) is driven solely by the *values* of its operands (and, for a store or load, by μ , which the relation keeps equal). An `ASSIGN-OP` step therefore writes the same word w on both sides; a `STORE` writes the same word to the same address, preserving $\mu = \mu'$; `BR-T/BR-F` branch the same way because the condition value matches; `JMP` and `CALL` are unaffected by operand renaming except that `CALL` passes equal argument words. In each case the single source step is matched by a single target step to τ' with $\tau \approx \tau'$: memories and call stacks stay equal, the pointers stay corresponding, and clause (3) is re-established because both sides updated the same destination (when present) to the same value, and the live set shrank or stayed the same.

Terminals (clause 3). A terminal σ on `ret v` with empty κ is matched by σ' on `ret  $A^*(v)$` ; by clause (3), $\llbracket v \rrbracket_{(\rho, \beta)} = \llbracket A^*(v) \rrbracket_{(\rho', \beta)}$, so both observe the same word. A bounds-trap terminal is reached on the target exactly when reached on the source, since a checked `addr_index` traps on the value of its index operand, which the relation keeps equal. Either way the observations agree.

All three clauses hold, so by [Theorem G.8](#), $\text{obs}(\mathcal{P}_{\text{copy}}(P)) = \text{obs}(P)$, and by [Theorem G.12](#) termination and trapping behaviour are preserved as well. $\square$

Implementation pointers

The pass lives in `CopyPropagationPass`; `propagate` drives one block, `identitySource` defines the recognized identity copies, `resolveAlias` computes A^* (with a 64-step guard against pathological chains, which a well-formed acyclic alias map never reaches), and the uniform operand substitution is delegated to `IrValueRewriter`. Two honest notes on the gap between code and theorem. First, the pass is purely intra-block: A is reset at every block boundary, which is why clause (3) of [Theorem G.49](#) is stated over live temporaries *within* a block; cross-block copy propagation is the job of the function-level `GlobalValuePropagationPass` ([Section G.9](#)). Second, the single-use guard is what makes the deletion sound without a separate liveness pass: a copy used twice is left in place (only its operands are renamed), so the proof never needs to reason about a deleted definition with surviving uses.

G.11 Pass: Dead Temp Elimination

`DeadTempEliminationPass` deletes assignments whose result is never read. Of all the passes this is the one whose correctness feels most self-evident — if nobody reads tn , surely it does not matter how tn was computed — and yet the obvious argument is wrong. “Nobody reads it” is not enough: the deleted right-hand side might *do* something on the way to producing its value. A load can fault, a call can store to memory or recurse, an increment mutates the slot it names. Throwing away such an instruction because its *result* is dead would also throw away its effect, and the observation would notice. The pass therefore deletes an assignment only when its right-hand side is *pure*, and the work of this proof is to pin down that side condition exactly and show it is the one the implementation actually tests.

What the pass does

For each block B the pass runs a backward liveness sweep over the temporaries. It seeds a live set with the temporaries read by the terminator (`IrUsageAnalyzer.usedTemps` on the terminator) and walks B ’s instructions from last to first. At an assignment $tn = r$ it asks two questions: is tn *not* in the current live set, and is r *pure*? If both hold, the assignment is dropped; otherwise tn is killed from the live set, the temporaries read by r are added, and the instruction is kept. Every non-assignment instruction (a store, a void call) is always kept and its read temporaries added to the live set. The purity test is `IrUsageAnalyzer.isPure`, which classifies exactly three right-hand-side shapes as *impure* — load, value call, and the in-place `incdec` form — and every other shape (the `addr_*` family, `binop`, `cmp`, `unop`, `cast`, and a constant) as *pure*.

A representative rewrite, from a redundant address computation left behind by an earlier pass:

before	after
<code>t4 = add t2, #1:int32 : int32</code>	
<code>t5 = mul t2, t3 : int32</code>	<code>t5 = mul t2, t3 : int32</code>
<code>ret t5</code>	<code>ret t5</code>

Here `t4` is read by no later instruction and no terminator, and its right-hand side `add` is pure, so the assignment to `t4` is deleted. By contrast, were the first line `t4 = load t2 : int32`, the pass would keep it even though `t4` is dead, because a load is impure.

Two facts the analysis relies on

The backward sweep is purely local to one block, yet it claims to establish that the deleted assignment’s result is globally unread. Two structural facts close that gap; both are inherited from the framework’s account of the IR (Section G.1).

Lemma G.51 (Temporaries are block-local). *In a well-formed program (Theorem G.1) every use of a temporary tn lies in the same block as its dominating definition. Equivalently, a temporary defined in block B is read only within B .*

Proof. The framework records (Section G.1) that temporaries are single-assignment within a block and that values cross block boundaries only through memory slots, never through temporaries: “a value reaches memory only through a store and returns only through a load.” The lowering that produces the IR (Chapter 6) materializes every value that must outlive a block into a slot; a temporary is therefore a within-block name. Hence a read of tn in a block other than its definition’s would have no dominating definition on the entry path, violating the definition-dominance clause of Theorem G.1. $\square$

Lemma G.52 (Pure right-hand sides have no effect on memory or control). *Let r be a pure right-hand side in the sense of `IrUsageAnalyzer.isPure`. Then the ASSIGN-OP step for $\text{tn} = r$ takes $\langle \pi, (\rho, \beta), \mu, \kappa \rangle \rightsquigarrow \langle \pi^+, (\rho[\text{tn} \mapsto w], \beta), \mu, \kappa \rangle$ with $w = \llbracket r \rrbracket_{\varphi, \mu}$ defined, and it changes only the binding of tn : the memory μ , the slot bases β , the call stack κ , and every temporary other than tn are left identical.*

Proof. By cases on the pure shapes. A constant, a binop, a `cmp`, a `unop`, and a `cast` evaluate $\llbracket r \rrbracket_{\varphi, \mu}$ as a total integer operation over operand values read from ρ and constants (Theorem G.2 makes division and modulus total, so even those pure operators never trap), reading μ for nothing and writing it not at all. The `addr_of_symbol`, `addr_field`, and `addr_index` forms compute an address from β and from operand values; address computation does not consult or modify μ . The single subtlety is the bounds-checked `addr_index` of Theorem G.2, which can raise the bounds trap. The implementation does not classify `addr_index` as impure, but it is also not a problem here: a trapping `addr_index` does not write μ , β , or κ , so the only step that can occur is exactly the ASSIGN-OP step above (when the index is in bounds) or a trap (when it is not), and the trapping case is handled by Theorem G.55 below rather than by this lemma. Excluded by purity are precisely the three forms that touch more than $\rho(\text{tn})$: `load` reads μ (and so its value depends on memory the deletion would no longer realize in step order, though the read itself is harmless), `value call` enters a callee that may store, recurse, or trap, and `incdec` writes μ . None of these is deleted. $\square$

The simulation relation

Fix a well-formed P and let $P' = \mathcal{P}(P)$ be its image under `DeadTempEliminationPass`. The two programs differ only by deleted pure assignments inside blocks; their globals, struct types, slot tables, block labels, terminators, and surviving instructions coincide.

Because temporaries are block-local (Theorem G.51), once control leaves a block the only state that matters downstream is (μ, β, κ) plus the slot contents in μ — never a temporary. Within a block, a deleted assignment’s destination is dead, hence read by nothing before it

is either overwritten or the block ends. We relate states that agree everywhere a surviving instruction or a terminator can look.

Definition G.53 (Dead-temp simulation). Define $\sigma \approx \sigma'$ between states of P and P' to hold when:

1. the memories agree, $\mu = \mu'$, as do the slot bases $\beta = \beta'$ and the call stacks κ, κ' (related componentwise by the same relation on each saved frame);
2. the two program pointers denote “the same place,” i.e. the same function and block and the same surviving instruction — a position pointing at a deleted assignment on the P side is identified with the position the P' side has already advanced past; and
3. the temporary environments agree on every temporary that is *live* at the current point: for all tm live at π , $\rho(tm) = \rho'(tm)$.

The anti-stuttering measure ([Theorem G.9](#)) $\mathfrak{m}(\sigma, \sigma')$ is the number of deleted assignments remaining between the P -pointer and the next surviving instruction; it is bounded by the block length and strictly decreases on each deleted-assignment step (the only stuttering step below).

Clause (3) is the crux: it deliberately says *nothing* about dead temporaries, which is exactly the freedom the pass needs.

Proof

Theorem G.54 (Semantic preservation of DeadTempEliminationPass). `DeadTempEliminationPass` is semantics-preserving in the sense of [Theorem G.6](#).

We discharge the obligation of [Theorem G.10](#): confirm well-formedness, then check the three simulation clauses of [Theorem G.7](#) for $\approx$.

Well-formedness ([Theorem G.11](#)). The pass deletes instructions but adds none, redirects no terminator, and touches no slot or type, so control-flow integrity and type agreement are immediate. The one obligation with content is definition dominance: deleting the definition of tn must not orphan a use. But the pass deletes $tn = r$ only when tn is dead at that point, and by [Theorem G.51](#) every use of tn is in this same block; a backward live set seeded from the terminator marks tn live at the deletion point iff some later instruction or the terminator reads it. Deadness therefore means no surviving use exists anywhere in P' , so $\vdash P' \text{ ok}$.

Clause (1): initial states. $\text{init}(P)$ and $\text{init}(P')$ enter main's entry block with empty temporary environment, identical zero-initialized memory and slot bases, and empty call stack. They agree on μ , β , κ , denote the same place, and agree on all (vacuously many) live temporaries, so $\text{init}(P) \approx \text{init}(P')$.

Clause (2): steps preserve $\approx$. Suppose $\sigma \approx \sigma'$ and $\sigma \rightsquigarrow \tau$. By [Theorem G.4](#) the step is forced by the instruction or terminator at σ 's pointer.

Deleted assignment (stutter). If σ 's pointer is at a pure assignment $tn = r$ that the pass deleted, then tn is dead at this point. The source step is the ASSIGN-OP of [Theorem G.52](#): it changes only $\rho(tn)$ and advances the pointer. Match it on the P' side with *zero* steps. The target state σ' is unchanged; we must show $\tau \approx \sigma'$. Memory, bases, and call stack are untouched by the source step, so clause (1) survives. The pointers still denote the same place by the identification in clause (2) of [Theorem G.53](#). For clause (3): the live set at τ is the live set at σ with tn removed and r 's reads added — but tn was dead, so it was not in the live set to begin with, and the temporaries r reads were already live before the assignment in the original program's liveness and unchanged in value here. Every temporary live at τ was live at σ (no new live temporary is introduced by removing a dead def) and has the same value in ρ and ρ' . Hence $\tau \approx \sigma'$. The measure $\mathfrak{m}$ strictly decreases, ruling out infinite stutter.

Surviving instruction or terminator (lock-step). Otherwise the form at σ 's pointer is present identically in P' . By [Theorem G.53](#) the two states agree on μ , β , κ , and on every *live* temporary; and the operands an instruction or terminator reads are, by definition of the live set, live at this point. So $\llbracket \cdot \rrbracket$ evaluates each operand to the same value on both sides, the same rule fires, and $\sigma' \rightsquigarrow \tau'$ with τ' updating the same component the same way:

- a *surviving (kept) assignment* $tm = r$ binds tm to the same w on both sides (its reads are live, hence equal), preserving clause (3); kept impure forms such as a load, a value call, or an incdec additionally agree on their memory/call-stack effect because $\mu = \mu'$, κ related;
- a *store* writes the same value to the same address in equal memories, keeping $\mu = \mu'$;
- a *void call* enters the same callee with equal arguments, pushing related return contexts;
- a *branch, jump, or return* reads a live condition or value equal on both sides and transfers control identically; for RET-CALLER the returned word is equal and lands in the related caller frame.

In each case $\tau \approx \tau'$; the live set is recomputed identically on both sides (same surviving instruction), so clause (3) is restored.

Clause (3): related terminals agree. A terminal state is a top-level `ret` v with empty call stack or the bounds trap. If σ is a top-level return, its pointer is at a `ret`, which survives in P' ; its operand v is live (it is read by the terminator), so $\llbracket v \rrbracket_\varphi = \llbracket v \rrbracket_{\varphi'}$, and σ' returns the same word, observed identically. If σ is the bounds trap, σ' reaches it too, by the following lemma, and both observe -6 .

Lemma G.55 (Deletion never removes a trap). *The only trapping reduction is a bounds-checked `addr_index` (Theorem G.2). The pass deletes such an assignment only if its result is dead. Even so the trap is preserved, because the trap is part of `obs` and must fire on both sides.*

Proof. This is the one place where the implementation’s purity classifier and the trapping semantics could disagree, so we are explicit. A dead, in-bounds `addr_index` is genuinely effect-free and its deletion is covered by the stutter case above. A dead *out-of-bounds* `addr_index`, however, traps in P while its deletion would let P' run on — a divergence the proof must not permit. We resolve it by observing that the FRISCC lowering never emits a checked `addr_index` as a dead assignment in the first place: the index computation that feeds a subscript is always consumed by the load or store it addresses, so its destination temporary is live by construction, and the pass’s deadness test fails. Under this invariant the deletion branch is reached only for in-bounds-safe pure forms, and the trap of a live, kept `addr_index` fires identically on both sides because $\beta = \beta'$ and the index operand is live and equal. Were a future front end to emit a dead checked subscript, soundness would require adding `addr_index` to `isPure`’s impure set; we record this as the pass’s standing precondition. $\square$

All three clauses hold, so by Theorem G.8 $\text{obs}(P') = \text{obs}(P)$, and Theorem G.54 follows. $\square$

Implementation pointers

The pass is `DeadTempEliminationPass` in `compiler-opt`; its `eliminate` method is the backward sweep of Theorem G.53’s measure, seeding the live set from `IrUsageAnalyzer.usedTemps(block.terminator())`. The purity classifier is `IrUsageAnalyzer.isPure`, whose impure set is exactly `{Load, Call, IncDecOp}` — matching Theorem G.52. The pass operates one block at a time and never inspects other blocks, which is sound precisely by Theorem G.51. The single divergence from the theorem’s clean statement is the checked-subscript caveat of Theorem G.55: the implementation classifies `addr_index` as pure and relies on the lowering invariant that a checked subscript’s address temporary is always live. This is a real precondition, not a paper-over; the classifier is correct only under it. The framework’s standing obligations (Theorem G.11, Theorem G.12) are discharged above and, for pipelines, by Theorem G.13. The classical treatment of dead-code elimination by liveness is Aho et al. [2] and Muchnick [62]; the discipline of restricting deletion to effect-free computations is the same one CompCert formalizes [56].

G.12 Pass: Dead Slot Store Elimination

`DeadSlotStoreEliminationPass` removes a store to a frame slot whose written value is overwritten before anyone reads it. If a slot s holds the result of store a, v and the next access to s on every path is another store — never a load — then the value v landed in memory and was buried there, observed by nothing. Deleting the first store cannot change the observation, because the observation can only ever see the *last* value written before a read. The delicacy is twofold. First, “overwritten before read” is a property of all forward paths, so the pass needs a backward dataflow analysis over the control-flow graph, not a single block scan. Second, the analysis only reasons about slots it can name precisely; an access through an address it cannot resolve, or any call, must be treated as possibly reading *every* slot, or the pass would delete a store whose value some opaque access later observes.

What the pass does

The pass computes, for each block, the set of slots that are *live-out*: slots whose current contents may be read along some path before being overwritten. This is a backward, may-analysis — liveness for memory slots rather than for temporaries. The transfer function (`transfer`) walks a block’s instructions from last to first over a live set initialized to the block’s live-out:

- a load from a resolvable slot s makes s live (its contents are about to be read); a load through an *unresolvable* address conservatively makes *all* tracked slots live;
- a value call or an incdec likewise makes all tracked slots live (the callee may read any of them);
- a void call also makes all tracked slots live;
- a store to a resolvable slot s *kills* s — but first, if s is *not* live at that point, the store is dead and is deleted; a store through an unresolvable address is kept and kills nothing.

The block-level live-out is the union over successors of their live-in, iterated to a fixed point (`rewriteFunction`’s round-robin reverse-order sweep over all blocks, repeated until stable). Slot identity is recovered by `BlockAddressTracker`, which tracks which temporary currently aliases the address of which tracked slot, following `addr_of_symbol` and pointer-cast chains; the set of tracked slots is the function’s LOCAL and PARAM slots (`SlotAddressResolver.trackedSlots`).

A representative rewrite: a local x overwritten before reuse.

before

```
t1 = addr_of_symbol local:x
store t1, #0 : int32
store t1, #7 : int32
t2 = load t1 : int32
```

after

```
t1 = addr_of_symbol local:x
store t1, #7 : int32
t2 = load t1 : int32
```

The first store of $\#0$ is dead: x is overwritten by $\#7$ before the load, so x is not live at the first store, and it is removed. (For readability these illustrative snippets elide the inline type suffix on constant operands; the real IR text writes them as $\#0:\text{int32}$.)

The aliasing assumption, stated

The analysis is only as sound as its notion of which accesses touch a slot. We make the relied-upon assumption explicit.

Convention G.56 (Conservative slot resolution). The pass partitions every memory access into one of two cases via `BlockAddressTracker.resolveSlot` / `SlotAddressResolver`: either it resolves to a uniquely named tracked slot s , or it is *opaque*. The implementation guarantees:

1. **Resolution is exact, not approximate.** If an access resolves to slot s , then it touches the storage of s and of no other tracked slot; resolution follows only `addr_of_symbol` of a tracked LOCAL/PARAM symbol and `ptrcast` chains thereof, so the named slot is the actual target.
2. **Opacity is conservative.** Any opaque *read* the tracker cannot tie to a named slot — a load through a computed or cast-laundered address, an `incdec`, or any `call` — is assumed to potentially read every tracked slot, so the transfer makes them all live. An opaque *store* (a store whose address resolves to no tracked slot) is simply kept and mutates the live set not at all: it can only overwrite storage, never read it, so it can neither resurrect a deleted value nor justify keeping any other store alive. In neither case is the opaque access itself deleted.

Under these two guarantees, a slot marked not-live at a store has, on every forward path, no resolvable load of s and no opaque access before the next store to s or function exit.

[Theorem G.56](#) is the load-bearing assumption. Guarantee (1) is what lets us say a kept store to $s' \neq s$ does not resurrect s 's deleted value; guarantee (2) is what stops the pass from deleting a store whose value an opaque access later reads.

The simulation relation

Fix a well-formed P , let $P' = \mathcal{P}(P)$. The two programs differ only by deleted stores to tracked slots; everything else coincides. The deleted store's footprint is the bytes of one slot in μ . The analysis certifies those bytes are dead — rewritten before any read — so the two memories may legitimately disagree there.

Definition G.57 (Dead-store simulation). For a state σ of P , let $\text{LiveOut}(\pi)$ denote the set of slots the analysis marks live at σ 's program point π . Define $\sigma \approx \sigma'$ to hold when:

1. the program pointers denote the same place (same function, block, surviving program point), the slot bases $\beta = \beta'$ agree, the temporary environments $\rho = \rho'$ agree, and the call stacks κ, κ' are related componentwise;
2. the memories agree on every byte *except* possibly the storage of slots that are *not* in $\text{LiveOut}(\pi)$: for every address a that belongs to a slot $s \in \text{LiveOut}(\pi)$, or to no tracked slot at all, $\mu(a) = \mu'(a)$.

The anti-stuttering measure ([Theorem G.9](#)) counts deleted stores remaining between the P -pointer and the next surviving instruction; it is bounded by block length and falls on each deleted-store step.

Clause (2) says the two memories may diverge only on dead slot storage. Crucially the relation is keyed to the program point: as control advances and a slot becomes live again (it is about to be read, because a later store re-established it), the relation must already have the memories agreeing there. The analysis is exactly what guarantees this, as the load case below shows.

Proof

Theorem G.58 (Semantic preservation of `DeadSlotStoreEliminationPass`). *Under [Theorem G.56](#), `DeadSlotStoreEliminationPass` is semantics-preserving ([Theorem G.6](#)).*

We discharge [Theorem G.10](#).

Soundness of the analysis. The forward reading of the backward fixed point is the invariant we need.

Lemma G.59 (Dead slots are unread until re-stored). *Let the analysis mark slot s as not live at program point π (so $s \notin \text{LiveOut}$ for the store at π). Then on every control-flow path forward from π , the first access that could read s 's storage is preceded by a store to s (a redefinition); equivalently, no resolvable load of s and no opaque access occurs before s is overwritten or the function returns.*

Proof. The live-out sets are the least fixed point of the standard backward liveness equations: $\text{LiveOut}(B) = \bigcup_{B' \in \text{succ}(B)} \text{LiveIn}(B')$ and LiveIn obtained from LiveOut by the per-block transfer. The transfer adds s to the live set exactly when it meets, scanning backward, a resolvable load of s , and adds *all* tracked slots when it meets an opaque access ([Theorem G.56](#), guarantee 2); it removes s only at a store to s . By the standard soundness of backward may-liveness [[44](#), [66](#)], $s \in \text{LiveOut}$ at π iff some path from π reaches a use of s — a resolvable load or an opaque access — with no intervening redefining store. Contrapositively, $s \notin \text{LiveOut}$ means every forward path overwrites s (or exits) before any such use. Exact resolution (guarantee 1) ensures a store to $s' \neq s$ is not counted as a use or a redefinition of s , so the equations track s alone. $\square$

Well-formedness (Theorem G.11). The pass deletes store instructions only; it adds nothing, moves no block, redirects no terminator, and binds no temporary. Control-flow integrity and definition dominance are untouched (no temporary's definition is removed). Type agreement is trivially preserved. Hence $\vdash P' \text{ ok}$.

Clause (1): initial states. $\text{init}(P)$ and $\text{init}(P')$ enter `main` with identical empty ρ , identical zero-initialized μ over identical slot bases, and empty κ . The memories agree everywhere, so clause (2) of Theorem G.57 holds a fortiori, and the pointers coincide. Thus $\text{init}(P) \approx \text{init}(P')$.

Clause (2): steps preserve $\approx$. Suppose $\sigma \approx \sigma'$, $\sigma \rightsquigarrow \tau$, with the step forced (Theorem G.4) by the form at σ 's pointer.

Deleted store (stutter). If the pointer is at a store a, v that the pass deleted, then a resolves to a slot $s \notin \text{LiveOut}$. The source step writes $\llbracket v \rrbracket_\varphi$ into s 's storage, advancing the pointer; match it with zero target steps. We check $\tau \approx \sigma'$. Pointers still denote the same place; β , ρ , κ untouched. For memory: the source step changed only s 's storage. At τ the live-out is the live-out *after* this store, which still excludes s unless a later instruction re-reads it — but by Theorem G.59 any forward read of s is preceded by a redefining store, so the bytes the source just wrote are dead and will be overwritten before being read. Concretely, at the very next point s becomes live again (Theorem G.57 clause 2 needs agreement on s only then), it is because a surviving store to s executes on both sides first, restoring $\mu(a) = \mu'(a)$ for s 's bytes. Until then $s \notin \text{LiveOut}$ and disagreement on s is permitted. All other bytes were equal and are unchanged. So clause (2) holds and $\tau \approx \sigma'$; the measure falls.

Surviving instruction or terminator (lock-step). The form is present identically in P' , and $\rho = \rho'$, so every operand value agrees. The same rule fires; $\sigma' \rightsquigarrow \tau'$.

- A *kept store* to a resolvable slot s writes the same value $\llbracket v \rrbracket_\varphi$ to s 's bytes on both sides, making μ and μ' agree on s thereafter — precisely the moment s may re-enter the live set — and leaving all other bytes as they were. A kept store through an *opaque* address writes the same value $\llbracket v \rrbracket_\varphi$ to the same address on both sides — the operands live in $\rho = \rho'$, so the value and target address agree. Wherever the two memories already agreed (every live and every untracked byte), they continue to agree after the identical write; wherever they were permitted to disagree (dead-slot storage), the write either leaves that byte untouched or overwrites it identically on both sides. Either way clause (2) is undisturbed: an opaque store reads nothing, so it neither demands fresh agreement nor introduces new disagreement.
- A *load* of slot s reads s 's bytes; the analysis marks s live at this point, so by Theorem G.57 clause (2) the memories agree on s , and the loaded word is equal, binding the destination temporary identically. (This is the case the whole relation is engineered to make true: a live slot is a slot the memories agree on.)

- A *value* or *void call* enters the same callee with equal arguments; the analysis made all tracked slots live before the call, so the memories agree on every slot the callee could read, and on all untracked bytes, hence the callee runs identically and returns equal results into related frames.
- A *branch*, *jump*, or *return* reads $\rho = \rho'$ conditions/values and transfers control identically.

In every case $\tau \approx \tau'$, the live-out advancing identically on both sides because the surviving program is the same.

Clause (3): related terminals agree. A top-level *ret* v reads its operand from $\rho = \rho'$, so both return the same word. The return value is a temporary or constant, never a slot read, so it is independent of any dead-slot disagreement. The bounds trap is reached identically: a checked `addr_index` is an assignment, untouched by this pass, and its operands live in $\rho = \rho'$. Both observe the same value or the same trap -6 .

All three clauses hold, so by [Theorem G.8](#) $\text{obs}(P') = \text{obs}(P)$, proving [Theorem G.58](#). $\square$

Implementation pointers

The pass is `DeadSlotStoreEliminationPass` in `compiler-opt`. Its `rewriteFunction` runs the backward round-robin dataflow iteration over per-block live-in/live-out sets; transfer is the per-block transfer function of [Theorem G.59](#), with `removeDeadStores` toggling between the analysis sweep and the rewriting sweep. The `IrVoidCallInstr`, `value Call`, and `IncDecOp` arms that `live.addAll(trackedSlots)` are guarantee (2) of [Theorem G.56](#) in code; the unresolvable-load arm (`slot == null` adding all tracked slots) is the same conservatism for opaque reads. Slot identity comes from `BlockAddressTracker` and the `LOCAL/PARAM` filter of `SlotAddressResolver.trackedSlots`, which is exactly the set the analysis is sound over.

One honest divergence from the cleanest possible statement: the implementation tracks *only* `LOCAL` and `PARAM` slots, never globals or spill slots, and resolves addresses only through `addr_of_symbol` and `ptrcast` chains. A store the tracker cannot tie to a tracked slot is simply kept (the `slot == null` arm), so the pass is conservative by construction: it may miss dead stores, but it never deletes a live one. This is the safe direction, and it is why [Theorem G.56](#) is an assumption the implementation *discharges* rather than one it merely hopes for. The standing obligations ([Theorem G.11](#), [Theorem G.12](#)) and pipeline composition ([Theorem G.13](#)) are inherited from the framework. The backward-liveness machinery is standard Kildall-style dataflow [44, 66]; the memory-aware variant and its alias caveats follow Muchnick [62] and the dead-store treatment of Cooper and Torczon [18].

G.13 Pass: Load Forwarding

`LoadForwardingPass` replaces a load from a slot by the value most recently stored to that slot, when no access in between could have changed the slot’s contents. If the program writes #7 to slot x and then, with nothing disturbing x , loads x into tn , the load is redundant: tn must equal #7. The pass forwards the stored value directly, turning a memory round-trip into a register-level copy and exposing the result to constant folding and copy propagation downstream. This is store-to-load forwarding, the dual of dead-store elimination: that pass removes a write nobody reads, this one removes a read whose answer is already known.

The entire correctness argument rests on one question — *can anything between the store and the load have changed the slot?* — and the pass answers it with a deliberately conservative alias analysis. If *any* access it cannot prove disjoint from the slot intervenes, the pass forgets the slot’s known value and forwards nothing. The proof makes that guard explicit and shows that under it the forwarded value provably equals the loaded value in every reachable state.

What the pass does

The pass maintains, per slot, a *known value*: either a constant or a temporary holding the value last stored. It runs a forward dataflow analysis carrying a map from tracked slots to known constants across blocks (`transferConstants`, merged at join points by keeping only constants agreed by all predecessors, `mergeIncomingConstants`), and then a forward rewrite within each block (`rewriteBlock`) that also tracks temporary-valued knowns locally. On encountering a load $a : \tau$ where a resolves to slot s and s has a compatible known value k , it rewrites the load’s right-hand side to k :

- if k is a constant whose type matches the load type, the load becomes that constant (`ConstRhs`);
- if k is a temporary tm of type `int32` and the load is `int32`, the load becomes `add tm, #0` — a copy that later passes clean up.

The known-value map is invalidated aggressively: a store to a slot s sets s ’s known value (or clears it if the stored value is not a clean constant or same-typed temporary); a store through an *unresolvable* address, a value call, an `incdec`, or a void call *clears the entire map* (`slotValues.clear()`); and binding a temporary invalidates any known value that referred to it (`invalidateValuesUsingTemp`). Slot identity is recovered through `addr_of_symbol` and `ptrcast` alias tracking (`trackAddressAlias`, `SlotAddressResolver`) over the function’s `LOCAL/PARAM` slots; cross-block forwarding is restricted to constants to avoid dangling temporary references.

A representative rewrite:

before

```
t1 = addr_of_symbol local:x
store t1, #7 : int32
t2 = load t1 : int32
```

after

```
t1 = addr_of_symbol local:x
store t1, #7 : int32
t2 = #7 : int32
```

The store to x establishes the known constant $\#7$; nothing intervenes, so the load is forwarded to $\#7$.

The aliasing guard, stated

Convention G.60 (No intervening aliasing access). The pass forwards a known value k for slot s to a load of s only when, since k was established for s , every executed instruction is one of:

1. a store to a slot s' *exactly resolved* and distinct from s (which cannot touch s , by exact resolution as in [Theorem G.56](#));
2. an instruction with no memory write at all (a pure assignment, a branch);

and *no* instruction of the following kinds has executed: a store through an address not exactly resolved to a slot distinct from s , a value or void call, or an incdec. Any such excluded instruction forces the pass to clear s 's known value (`slotValues.clear()` for the global cases, or a targeted overwrite for a resolvable store to s itself), so a forward only survives when guarantee (1)/(2) held throughout. Equivalently: a forwarded load is guarded by the absence of *any* access that could have written s 's storage between the defining store and the load.

[Theorem G.60](#) is the assumption the proof rests on, and the implementation's clearing discipline is exactly its enforcement: every operation whose effect on s the analysis cannot bound resets s 's known value, so a surviving known value is a certificate that s was untouched.

The simulation relation

Fix a well-formed P , let $P' = \mathcal{P}(P)$. The two programs differ only in that some loads in P have become a constant or a copy in P' . No store, terminator, call, or address computation changes; memory is written identically on both sides. The only question is whether a rewritten load produces the same value as the original.

Definition G.61 (Load-forwarding simulation). Define $\sigma \approx \sigma'$ to hold when the two states are *identical* in every component: same program pointer (same place), $\mu = \mu'$, $\beta = \beta'$, $\rho = \rho'$, and componentwise-related κ, κ' . The anti-stuttering measure is identically zero: the simulation is strict lock-step, with each source step matched by exactly one target step ([Theorem G.9](#) discharged vacuously).

This is the tightest possible relation, and we can afford it because load forwarding deletes nothing and reorders nothing: it only changes *how* one temporary’s value is computed, not the value itself. The content of the proof is entirely in showing the rewritten load lands the same word.

Proof

Theorem G.62 (Semantic preservation of LoadForwardingPass). *Under Theorem G.60, LoadForwardingPass is semantics-preserving (Theorem G.6).*

The whole burden reduces to the value-equality lemma below; given it, the simulation is immediate.

Lemma G.63 (The known value equals memory). *Suppose the analysis holds known value k for slot s at a program point π , and let $\sigma = \langle \pi, \varphi, \mu, \kappa \rangle$ be any reachable state at π . Then $\llbracket k \rrbracket_\varphi = \text{load}_\tau(\mu, \text{addr}_\beta(s))$ for the load type τ at which the forward is performed; that is, the known value reads back exactly the word currently stored in s .*

Proof. By induction on the length of the run reaching σ , following how k entered the known-value map and showing the invariant “ k ’s value equals s ’s memory contents” is maintained from that point to π .

Establishment. k is set for s only by a store $\text{store } a, v : \tau$ with a exactly resolved to s ($\text{normalizeStoredValue}$). The STORE rule of the framework writes $\text{store}_\tau(\mu, \text{addr}_\beta(s), \llbracket v \rrbracket_\varphi)$, so immediately after the store, s ’s memory holds $\llbracket v \rrbracket_\varphi$. The recorded k is either the constant v itself (when v is a constant, normalized to the store type — normalizeConstant matches the type so the stored and recorded constants denote the same word) or the temporary $\text{tm} = v$ when v is a same-typed temporary. In both cases $\llbracket k \rrbracket_\varphi = \llbracket v \rrbracket_\varphi$ = the just-stored word. The base of the induction holds.

Preservation across intervening steps. Consider any step between establishment and π . By Theorem G.60, while k remains recorded for s the only steps that occur are stores to slots exactly resolved and distinct from s , and instructions with no memory write.

- A store to $s' \neq s$ (exactly resolved) writes only s' ’s bytes; by exact resolution it does not overlap s ’s storage, so $\text{load}_\tau(\mu, \text{addr}_\beta(s))$ is unchanged. It may set or clear s' ’s known value, but leaves s ’s.
- A pure assignment $\text{tj} = r$ changes only $\rho(\text{tj})$ and never μ . If k is the temporary tm and this assignment *rebinds* tm , the implementation invalidates k ($\text{invalidateValues UsingTemp}$), so k no longer holds for s and the invariant is not claimed past this point. Otherwise $\llbracket k \rrbracket_\varphi$ is unchanged (k is a constant, or a temporary not rebound), and s ’s memory is unchanged, so the equality persists.
- A branch or jump changes only the pointer.

Any step *outside* this list — an opaque store, a call, an incdec — would, by [Theorem G.60](#), have cleared s ’s known value; since k is still recorded at π , no such step occurred. Therefore the equality $\llbracket k \rrbracket_\varphi = \text{load}_\tau(\mu, \text{addr}_\beta(s))$ holds at π .

For a value k that crossed a block boundary, the cross-block analysis forwards only *constants* agreed by every predecessor (`mergeIncomingConstants`). A constant’s value is frame-independent ($\llbracket \#c \rrbracket_\varphi = c$), and the merge keeps k for s only if every predecessor’s exit recorded the same constant for s under the same clearing discipline; so along whichever path reached π , s ’s memory holds that constant by the single-block argument applied to that path. (Temporaries are never forwarded across blocks, matching [Theorem G.51](#) and the pass’s own comment that cross-block temp forwarding would dangle.) $\square$

Well-formedness ([Theorem G.11](#)). The pass replaces one right-hand side with another of the *same result type*: a constant k is forwarded only when its type equals the load type (`toForwardedRhs` checks that the constant’s type equals the `loadType`), and a temporary k only when both are `int32`, producing an `int32 add tm, #0`. So type agreement holds. No block, terminator, or definition is added or removed, so control-flow integrity and definition dominance are preserved (the forwarded temporary tm is, by construction, defined before the load it replaces — it is the stored value’s temporary, established earlier in the same block). Hence $\vdash P' \text{ ok}$.

Clauses (1)–(3) of the simulation. Clause (1): $\text{init}(P) = \text{init}(P')$ componentwise, so they are $\approx$ -related. Clause (2): suppose $\sigma \approx \sigma'$, i.e. $\sigma = \sigma'$, and $\sigma \rightsquigarrow \tau$. The instruction at the pointer is identical in P and P' *except* possibly a forwarded load. If it is not a forwarded load, the same rule fires on identical states, giving $\tau = \tau'$, so $\tau \approx \tau'$. If it is a forwarded load — P has $tn = \text{load } a : \tau$ where a resolves to s with known value k , and P' has $tn = k'$ where k' is the forwarded right-hand side (`#c` or `add tm, #0`) — then both are `ASSIGN-OP` steps on tn . The source binds $tn \mapsto \text{load}_\tau(\mu, \text{addr}_\beta(s))$. The target binds $tn \mapsto \llbracket k' \rrbracket_\varphi$, which is c for the constant case and $\rho(tm) + 0 = \rho(tm) = \llbracket tm \rrbracket_\varphi$ for the copy case. By [Theorem G.63](#), $\llbracket k \rrbracket_\varphi = \text{load}_\tau(\mu, \text{addr}_\beta(s))$, and k' denotes the same word as k (the constant is k ; the copy denotes the recorded temporary’s value). Hence both bind tn to the same word, and since all other state is unchanged identically, $\tau = \tau'$, so $\tau \approx \tau'$. Clause (3): related terminals are *equal* states, so they trivially agree on observation.

By [Theorem G.8](#), $\text{obs}(P') = \text{obs}(P)$, proving [Theorem G.62](#). $\square$

Implementation pointers

The pass is compiler-opt. `rewriteBlock` performs the forward rewrite described above, holding a map from each slot to a known constant-or-temporary `KnownValue`; `transferConstants` plus `mergeIncomingConstants` compute the cross-block constant facts the inductive step of [Theorem G.63](#) relies on. The clearing calls — `slotValues.clear()` on an unresolvable store, any `Call`, `IncDecOp`, or void call, and `invalidateValuesUsingTemp` on a temporary rebinding

— are [Theorem G.60](#) enforced in code. `toForwardedRhs` is the type-guarded rewrite that keeps [Theorem G.11](#)’s type clause true.

Two honest divergences from the cleanest theorem. First, temporary forwarding is *intentionally block-local*: the pass forwards a temporary-valued known only within one block and downgrades to constants-only across joins, because a temporary from a predecessor block has no defining occurrence on the path and would dangle (the pass’s own source comment). This is a soundness measure, not an oversight, and it is why [Theorem G.63](#)’s cross-block case is restricted to constants. Second, the temporary-copy forward emits `add tm, #0` rather than a bare copy, relying on a later copy-propagation/constant-folding pass to collapse it; the value is identical ($x + 0 = x$ in the IR’s modular int32 arithmetic), so this is semantically transparent but leaves a cosmetic instruction the pipeline removes. The standing obligations ([Theorem G.11](#), [Theorem G.12](#)) and composition ([Theorem G.13](#)) are inherited. Redundant-load elimination and store-to-load forwarding are standard; see Muchnick [62] and Cooper and Torczon [18], with the alias-soundness discipline matching the memory model of CompCert [56].

G.14 Loop-invariant code motion

The shared framework of [Section G.1](#) fixes the IR, its small-step semantics $\rightsquigarrow$, the observation `obs`, the simulation discipline of [Theorem G.7](#), and the single proof obligation [Theorem G.10](#). This section discharges that obligation for FRISCCc’s loop-invariant code-motion pass. The classical account of the optimization is in Aho et al. [2, §9.5] and Muchnick [62, §13.2]; what we prove here is the correctness of the conservative *intra-block* form the FRISCCc pass actually implements.

What the pass does

The implementation is `LoopInvariantCodeMotionPass` in `compiler-opt`. It first identifies the set of *loop blocks* of each function by the standard back-edge test (`detectLoopBlocks`): a back edge runs from a tail block to a header that precedes it or equals it in block order (the single-block self-loop, where header and tail coincide, is admitted too), and the natural loop is the header together with every block that reaches the tail without leaving through the header (`naturalLoop`). For each loop block — and *only* within that single block — the pass then reorders instructions (`reorderInvariantAssignments`).

Inside one block it computes the set of *invariant* assignment indices by a least-fixed-point sweep. An assignment `tn = r` is invariant when (i) its right-hand side `r` is *pure* — `isPure` rejects exactly `load`, `call`, and the increment/decrement form, admitting every other right-hand side — and (ii) every temporary `r` reads is either defined nowhere in this block or defined by another instruction already marked invariant. The sweep repeats until no new index is added, so the marked set is closed under this rule. The rewrite is then a single stable partition of the instruction list: every invariant instruction is emitted first, in its original

relative order, followed by every non-invariant instruction, in its original relative order; the terminator is untouched (`new IrBlock(block.label(), moved, block.terminator())`). The pass fires only when at least one invariant instruction originally sat *after* a non-invariant one (`hasMotionOpportunity`); otherwise the block is returned unchanged.

Two facts about the implementation shape the proof, and both are worth stating plainly because they make the argument far simpler than the textbook pass.

Remark G.64 (The pass never leaves the block). Despite the name, this pass does *not* hoist computations into a loop pre-header. It is a within-block reordering: the moved instructions stay in the same block, bracketed by the same terminator, and are therefore executed on exactly the same set of runs as before — once per visit to the block, never on a run that skips the block. The “may-not-execute” hazard that dominates the correctness of true pre-header hoisting (an instruction lifted above the loop test runs even when the loop body runs zero times) *does not arise here*, because nothing crosses a block boundary. The class comment records this design choice explicitly: motion is kept intra-block “to keep IR validity under current verifier constraints.” Purity still does the real work, as the next remark explains.

Remark G.65 (Why purity is the load-bearing hypothesis). Reordering two instructions within a straight-line block is sound precisely when neither depends on an effect of the other. The pass guarantees this in two layers. First, an invariant instruction reads no temporary defined by a *non*-invariant instruction (clause (ii)), so moving it earlier never moves it above a definition it consumes. Second, its right-hand side is pure (clause (i)): it performs no load and no call, so it neither reads memory that an earlier store might change nor writes memory that a later load might observe. The only state a pure right-hand side touches is the temporary environment, and it touches that state only by writing its own destination. Hoisting it over a non-invariant instruction therefore commutes: the two instructions write disjoint temporaries and share no memory effect.

The simulation relation

Fix a well-formed P (Theorem G.1) and let $P' = \mathcal{P}(P)$ be its image under the pass. If the pass reports no change, $P' = P$ and there is nothing to prove, so assume it rewrote the block ℓ^* of function f^* by the stable partition above. Write $B = I_0 I_1 \dots I_{m-1}$ for the original instruction list of ℓ^* and $B' = \pi(B)$ for the permuted list, where π is the stable partition that brings the invariant indices to the front. Let $\text{Inv} \subseteq \{0, \dots, m-1\}$ be the marked invariant set.

Both programs run the same instruction *multiset* in block ℓ^* ; they differ only in the order, and only within that one block. We relate states by “same everywhere, and same accumulated effect of ℓ^* so far.” Concretely, for a P -state $\sigma = \langle \pi_\sigma, \varphi, \mu, \kappa \rangle$ and a P' -state $\sigma' = \langle \pi_{\sigma'}, \varphi', \mu', \kappa' \rangle$ define $\sigma \approx \sigma'$ to hold when one of the following cases applies.

Outside ℓ^* . The current program point of σ is not inside ℓ^* (it is in another block, or in another function up the call stack). Then $\approx$ requires $\pi_\sigma = \pi_{\sigma'}$, $\mu = \mu'$, $\kappa = \kappa'$, and $\rho = \rho'$ on every temporary.

Inside ℓ^* . Both pointers sit in ℓ^* , at instruction indices i (in B) and i' (in B') respectively, such that the *set of original instructions already executed* on entry to position i in B equals the set already executed on entry to position i' in B' . We require $\mu = \mu'$, $\kappa = \kappa'$, and $\rho = \rho'$ on every temporary that is *not* the destination of an instruction still pending in *both* runs.

The anti-stuttering measure (Theorem G.9) is $\mathbf{m}(\sigma, \sigma')$ = the number of original instructions of ℓ^* that σ has executed but σ' has not yet; it is bounded by m and strictly drops on every stutter, as the lemma below shows. The relation is initial-respecting trivially: the initial states share the empty call stack, the same fresh μ , the entry pointer of `main`, and the empty temporary environment, none of which the pass changes (Theorem G.7, clause 1).

The commutation lemma

The heart of the matter is that the reordering does not change the state reached at the end of the block.

Lemma G.66 (Stable partition preserves the block's effect). *Let block ℓ^* be entered in frame-memory state (φ, μ) . Running the original list B to its terminator and running the permuted list $B' = \pi(B)$ to its terminator yield identical frames, identical memories, and identical evaluations of the terminator's operands.*

Proof. B' is obtained from B by a finite sequence of adjacent transpositions, each swapping a non-invariant instruction I_j with an immediately following invariant instruction I_k ($k > j$, $k \in \text{Inv}$, $j \notin \text{Inv}$); the stable partition is exactly the bubble-sort that floats every invariant instruction past the non-invariant ones ahead of it while preserving relative order within each class. It suffices to show each such adjacent swap preserves the running state, after which composing the swaps preserves it overall.

Consider one swap of I_j (non-invariant) and $I_k = (\text{tn} = r)$ (invariant), executed from some intermediate state (φ_0, μ_0) .

Memory. I_k is pure (clause (i) of invariance), so by Theorem G.65 its ASSIGN-OP step leaves μ unchanged: $\llbracket r \rrbracket_{\varphi_0, \mu_0}$ is computed and written to `tn` in ρ , with no `store $\tau$` and no enter. Hence the memory after the pair $I_j; I_k$ and after the swapped pair $I_k; I_j$ is the same: it is whatever I_j alone produces from μ_0 , since I_k touches no memory in either order.

Temporaries. I_k reads only temporaries that are either undefined in this block or invariant (clause (ii)). The instruction I_j is non-invariant, so it is *not* among the definitions clause (ii) permits I_k to read; therefore I_k does not read I_j 's destination, and $\llbracket r \rrbracket_{\varphi_0, \mu_0}$ takes the same

value whether I_j has run or not. Conversely, I_j does not read tn : a single-assignment-within-a-block invariant (Section G.1) makes tn the unique in-block definition of that temporary, and were I_j to read it, I_j would read a value that in the original order I_k has not yet produced — impossible in a well-formed block, where every use is dominated by its definition. The two destinations are distinct (again by single-assignment), so the two writes commute. Either order leaves ρ identical on tn , on I_j 's destination, and on every other temporary.

Each adjacent swap thus preserves (φ, μ) ; the full permutation, a composition of such swaps, does too. In particular the terminator, which is identical in both blocks and reads only the final frame, sees the same operand values. $\square$

Discharging the simulation clauses

Proposition G.67 ($\approx$ is a simulation). *The relation $\approx$ of this section satisfies the three clauses of Theorem G.7.*

Proof. Clause (1) is the initial-state observation made above. For clause (2) take $\sigma \approx \sigma'$ and a source step $\sigma \rightsquigarrow \tau$.

Outside ℓ^ .* Every block other than ℓ^* is byte-for-byte identical in P and P' , as is every other function. By Theorem G.4 the unique applicable rule of Figure G.1 fires identically on both sides; the matching target step $\sigma' \rightsquigarrow \tau'$ reproduces it, and $\tau \approx \tau'$ in the same case (or, when the step is a `jmp/br into  $\ell^*$` , in the inside case with both runs at the start of their respective lists, having executed the empty set of ℓ^* -instructions). This is lock-step; $\mathbf{m}$ is irrelevant.

Inside ℓ^ .* The source executes the next original instruction I_i . By Theorem G.66 the permuted run, taken to the same “set of instructions executed” frontier, agrees on the resulting frame and memory restricted to already-settled temporaries. Concretely: if I_i is one whose image in B' has not yet been reached, the target stutters (zero steps) and we advance the source bookkeeping only — $\mathbf{m}$ strictly decreases because one more source instruction has been executed without a matching target step. If instead the frontier now calls for I_i on the target side too, the target takes the one matching step. In both sub-cases the invariant “same set executed, same μ and κ , same ρ off the pending destinations” is restored, because by the commutation lemma the order of execution does not change the cumulative effect. When the frontier reaches the terminator, both lists have executed the full multiset; the terminator step is then lock-step and lands both runs in the outside case (or back at ℓ^* 's head on a loop back-edge) with identical (φ, μ, κ) .

Clause (3): a terminal source state is a top-level `ret  $v$` or the bounds trap; in the outside case $\sigma' = \sigma$ on all observable components, so σ' is already terminal with the same observation. Inside ℓ^* a state is never terminal (a block ends in a terminator, handled by the outside-bound terminator step), so this clause is reached only outside, where it holds by equality. $\square$

Well-formedness and conclusion

The three sub-obligations of [Theorem G.11](#) hold by inspection. *Control flow*: the pass keeps each block’s label and terminator (`new IrBlock(label, moved, terminator)`) and adds, removes, or redirects no block, so every `br/jmp` target still names a block of the same function. *Definition dominance*: within the block the only definition of each `tn` is its single in-block assignment, and clause (ii) forbids moving an invariant instruction above a definition it reads while the stable partition never moves a definition below a use of it (a use is non-invariant whenever the definition is, or the use is invariant and stays after its invariant definition because relative order within `Inv` is preserved); uses in *later* blocks see the same final frame by [Theorem G.66](#), so cross-block dominance is untouched. *Type agreement*: no right-hand side, store, or cast is rewritten — instructions are only permuted — so every instruction type-checks exactly as before. Hence $\vdash P' \text{ ok}$.

Theorem G.68 (Semantic preservation of loop-invariant code motion). *For every well-formed program P , $\text{obs}(\mathcal{P}(P)) = \text{obs}(P)$, where $\mathcal{P}$ is `LoopInvariantCodeMotionPass`.*

Proof. If the pass reports no change the claim is trivial. Otherwise [Theorem G.67](#) exhibits a simulation with a well-founded anti-stuttering measure, and the preceding paragraph establishes $\vdash P' \text{ ok}$. By [Theorem G.10](#) and [Theorem G.8](#), $\text{obs}(P') = \text{obs}(P)$. Since the pass is applied per function and the per-block rewrites compose, repeated application across functions and across optimizer sweeps preserves the observation by [Theorem G.13](#). $\square$

Remark G.69 (Divergence is preserved exactly). The intra-block discipline makes the termination argument ([Theorem G.12](#)) immediate: the pass changes neither the number of times ℓ^* is entered nor the value its terminator branches on ([Theorem G.66](#)), so a loop that ran forever still runs forever and a loop that exited still exits on the same iteration. There is no risk — present in true pre-header hoisting of an impure or trapping computation — of making a previously skipped computation run, because nothing is lifted out of the loop body.

G.15 Induction-variable strength reduction

This section discharges the obligation of [Theorem G.10](#) for FRISCCc’s induction strength-reduction pass, against the framework of [Section G.1](#). Classical strength reduction replaces a multiplication of an induction variable by a constant with an addition maintained across the loop’s back edge (Aho et al. [2, §9.1], Muchnick [62, §14.1]). The FRISCCc pass implements a conservative, *local* variant of that idea: it never introduces a new loop-carried accumulator, and instead rewrites the multiply in place into a shift-plus-add that is algebraically equal to it on every iteration. That makes the correctness argument a pointwise arithmetic identity rather than a loop-carried invariant, and the proof below is correspondingly short.

What the pass does

The implementation is `InductionStrengthReductionPass` in `compiler-opt`. Its loop analysis (`analyzeLoops`, `inductionSlots`) finds the natural loops by the back-edge test and, within them, the local frame slots that behave as induction variables: a slot is recorded as *induction* when, inside some loop block, a temporary loaded from that slot is incremented by an integer constant and the incremented value is stored back to the same slot (`detectIncrement` followed by the store-back check). This identifies, for example, the `i` of a for-loop counter held in a frame slot.

The rewrite (`rewriteBlock`) is purely intra-block. Walking a loop block, the pass tracks which temporaries currently hold the address of a slot (`addr_of_symbol`) and which hold a value freshly loaded from an induction slot. When it then meets an assignment

$$td = \text{mul } ti, \#c : \text{int32} \quad (\text{or the commuted } \text{mul } \#c, ti),$$

whose result type is `int32`, whose `ti` is a value just loaded from an induction slot, and whose constant multiplier `c` is an `int32` constant in $\{3, 5, 9\}$ (`isSupportedMultiplier`; the pass also checks the constant operand’s own type is `IrPrimitiveType.INT32`), it replaces that single instruction with the pair

$$tf = \text{shl } ti, \#k : \text{int32}; \quad td = \text{add } tf, ti : \text{int32},$$

where `tf` is a *fresh* temporary (index $\geq \text{maxTemp} + 1$, computed once per function) and the shift amount `k` is 1, 2, or 3 for `c = 3, 5, 9` respectively (`shiftAmount`). Every other instruction, every terminator, and every non-matching multiply is copied verbatim.

Remark G.70 (The supported multipliers are exactly $2^k + 1$). The three admitted constants are not arbitrary: $3 = 2^1 + 1$, $5 = 2^2 + 1$, $9 = 2^3 + 1$, and the chosen shift amount is precisely that `k`. This is the entire arithmetic content of the rewrite, and the correctness lemma below is just the observation that $i \cdot (2^k + 1) = (i \ll k) + i$ over the machine-word arithmetic the interpreter uses.

Remark G.71 (Why “induction” is incidental to soundness). The loop and induction-variable analysis decides *whether* to fire the rewrite, not whether it is correct. Because the replacement is a local algebraic identity evaluated from the same operand `ti` in the same place, it would be sound for *any* `ti` and any `c` $\in \{3, 5, 9\}$, induction variable or not. The analysis exists only to target the multiplications that recur across iterations, where the payoff is real; we use the loop machinery to explain the pass’s intent but do not rely on it for the proof. In particular we need no loop-carried invariant of the form “the accumulator equals $i \cdot c$ on every iteration,” because the pass maintains no accumulator — it recomputes $(i \ll k) + i$ from the live `ti` each time the instruction is reached.

The shift–add identity

Lemma G.72 (Shift–add equals multiply). *Let the interpreter evaluate `int32` operations as the corresponding operations on machine words (Theorem G.2). For every word `v` and*

every $c \in \{3, 5, 9\}$ with k the unique exponent satisfying $c = 2^k + 1$,

$$\llbracket \text{shl } v, k \rrbracket + v = \llbracket \text{mul } v, c \rrbracket.$$

Proof. The interpreter's `shl` computes $v \ll k$, which on a two's complement machine word is $v \cdot 2^k \bmod 2^{32}$, and its `add` computes addition modulo 2^{32} ; its `mul` computes multiplication modulo 2^{32} . Working in the ring $\mathbb{Z}/2^{32}\mathbb{Z}$,

$$(v \ll k) + v = v \cdot 2^k + v = v \cdot (2^k + 1) = v \cdot c,$$

all equalities holding modulo 2^{32} because the ring operations distribute. The left-hand side is exactly what the emitted pair computes ($\text{tf} = v \ll k$ then $\text{td} = \text{tf} + v$), and the right-hand side is what the original `mul` computes. The identity holds for every v , including the overflowing and negative cases, since it is an identity in the ring itself, not merely on a restricted range. $\square$

The simulation relation

Fix a well-formed P and let $P' = \mathcal{P}(P)$. If the pass reports no change, $P' = P$. Otherwise it has replaced, in some loop block, one or more matching multiplies by their shift-add pairs, each pair introducing a fresh temporary `tf` that occurs nowhere else in the function (the fresh index is taken strictly above `maxTemp`).

We use a near-lock-step relation that allows the target to stutter by the single extra instruction the rewrite inserts. For a P -state $\sigma = \langle \pi_\sigma, (\rho, \beta), \mu, \kappa \rangle$ and a P' -state $\sigma' = \langle \pi_{\sigma'}, (\rho', \beta), \mu', \kappa' \rangle$, say $\sigma \approx \sigma'$ when:

- $\mu = \mu'$ and $\kappa = \kappa'$ (memories and call stacks coincide; the pass changes neither);
- the program points correspond under the obvious instruction map — identical outside any rewritten block, and inside a rewritten block the source index i corresponds to the target index obtained by expanding each preceding matched multiply into its two-instruction pair, with the source either *at* a matched multiply (target then poised at the `shl`, having not yet stepped it) or *not*;
- ρ and ρ' agree on every temporary except the fresh `tf` of any pair whose `shl` the target has executed but whose `add` it has not yet executed — a transient that ρ does not name and that no source or target instruction outside the pair reads.

The anti-stuttering measure ([Theorem G.9](#)) is the number of half-completed pairs, i.e. $\mathfrak{m} = 1$ exactly when the target has executed a `shl` whose paired `add` is still pending and $\mathfrak{m} = 0$ otherwise; it is bounded by 1 and drops to 0 on the `add` step. Initial states are related (clause 1): nothing in `main`'s entry is rewritten, μ and κ start identical, and $\rho = \rho'$ are both empty.

Discharging the simulation clauses

Proposition G.73 ($\approx$ is a simulation). *The relation $\approx$ satisfies the three clauses of Theorem G.7.*

Proof. Clause (1) holds as just noted. For clause (2), let $\sigma \approx \sigma'$ and $\sigma \rightsquigarrow \tau$, by cases on the source instruction.

An unrewritten instruction or terminator. Both programs hold the identical form at corresponding points. By determinism (Theorem G.4) the same rule of Figure G.1 fires on each side. Since $\rho = \rho'$ on every operand temporary (the only disagreement allowed is on a fresh $\mathbf{tf}$ whose pair is mid-flight, which an unrewritten instruction cannot name) and $\mu = \mu'$, the two steps produce frames and memories that again agree off the transient, and call-stack effects coincide. This is lock-step; $\tau \approx \tau'$ with $\mathbf{m}$ unchanged at 0.

A matched multiply $\mathbf{td} = \mathbf{mul} \ \mathbf{ti}, \#c$. The source takes one ASSIGN-OP step writing $w = \llbracket \mathbf{mul} \ \mathbf{ti}, \#c \rrbracket_{\varphi, \mu} = \llbracket \mathbf{ti} \rrbracket_{\varphi} \cdot c$ into $\rho(\mathbf{td})$. On the target the corresponding point is the $\mathbf{shl}$ of the pair. We match the single source step by the *two* target steps

$$\mathbf{tf} = \mathbf{shl} \ \mathbf{ti}, \#k \quad \text{then} \quad \mathbf{td} = \mathbf{add} \ \mathbf{tf}, \mathbf{ti}.$$

Because $\rho = \rho'$ agree on $\mathbf{ti}$ (it is not the transient of any other pair: the analysis matches a freshly loaded induction value, and the fresh $\mathbf{tf}$ indices are mutually distinct), the first target step writes $\llbracket \mathbf{ti} \rrbracket_{\varphi'} \ll k$ into the fresh $\mathbf{tf}$, and the second writes $(\llbracket \mathbf{ti} \rrbracket_{\varphi'} \ll k) + \llbracket \mathbf{ti} \rrbracket_{\varphi'}$ into $\mathbf{td}$. By Theorem G.72 that value is exactly $\llbracket \mathbf{ti} \rrbracket_{\varphi'} \cdot c = w$, the same word the source wrote to $\mathbf{td}$. Neither target step touches μ or κ (both are pure ASSIGN-OP instances). After the pair, ρ' equals ρ on $\mathbf{td}$ and on every prior temporary, differing only on the now-dead $\mathbf{tf}$, which the relation tolerates and which no later instruction reads. Hence $\tau \approx \tau'$ with $\mathbf{m} = 0$. (Choosing to match with two steps at once means the relation never actually rests in the half-completed state; the measure is there only to license the alternative bookkeeping in which one counts the $\mathbf{shl}$ and $\mathbf{add}$ as separate stutters, and it strictly decreases there too.)

Clause (3): a terminal source state returns at top level or traps; in either case the corresponding target point holds the identical terminator (terminators are never rewritten) over a frame agreeing on the returned operand — which is some $\mathbf{td}$ or constant, never a transient $\mathbf{tf}$ — so the target reaches a terminal with the same observation. $\square$

Well-formedness and conclusion

The sub-obligations of Theorem G.11 hold. *Control flow:* blocks, labels, and terminators are preserved verbatim (new IrBlock(label, rewritten, terminator)); no edge changes. *Definition dominance:* the inserted $\mathbf{tf}$ is fresh, defined by its $\mathbf{shl}$ exactly one instruction before its sole use in the paired $\mathbf{add}$, so its use is dominated by its definition; $\mathbf{td}$ keeps its single in-block definition, now produced by the $\mathbf{add}$ at the same position the $\mathbf{mul}$ occupied, so all downstream uses of $\mathbf{td}$ remain dominated. *Type agreement:* every emitted instruction is

typed `int32` — the guard requires `binOp.resultType() == IrPrimitiveType.INT32` and the constructors stamp `int32` on the `shl`, the shift constant, and the `add` — matching the `int32` result type of the multiply it replaces. Hence $\vdash P' \text{ ok}$.

Theorem G.74 (Semantic preservation of induction strength reduction). *For every well-formed program P , $\text{obs}(\mathcal{P}(P)) = \text{obs}(P)$, where $\mathcal{P}$ is `InductionStrengthReductionPass`.*

Proof. If the pass reports no change, immediate. Otherwise [Theorem G.73](#) furnishes a simulation with the bounded measure $\mathbf{m} \in \{0, 1\}$, and well-formedness holds by the paragraph above. [Theorem G.10](#) and [Theorem G.8](#) give $\text{obs}(P') = \text{obs}(P)$; [Theorem G.13](#) extends this across functions and optimizer sweeps. $\square$

Remark G.75 (On the loop-carried-invariant phrasing). A textbook strength-reduction pass introduces an accumulator a initialized to $i_0 \cdot c$ in the pre-header and stepped by c on each back edge, and its correctness rests on the loop invariant “ $a = i \cdot c$ at every iteration,” proved by establishing it in the pre-header and preserving it across the back edge. The `FRISCCc` pass deliberately does *not* do this — it would require cross-block temp motion the verifier presently disallows (cf. [Theorem G.64](#)) — so the invariant degenerates to the pointwise identity of [Theorem G.72](#), established afresh at each execution of the rewritten instruction. The stronger invariant is the right thing to prove only once the pass is generalized to a true loop-carried accumulator; until then this local argument is both sufficient and faithful to the code.

G.16 Common-subexpression elimination

This section discharges [Theorem G.10](#) for `FRISCCc`’s local common-subexpression elimination, using the framework of [Section G.1](#). The optimization is textbook (Aho et al. [2, §9.1, §9.2], Appel [7, §17.3]); the `FRISCCc` implementation is its *local* (single-block) form built on a value-numbering table, which keeps the available-expressions reasoning entirely within one straight-line block and so admits a direct simulation.

What the pass does

The implementation is `CommonSubexpressionEliminationPass` in `compiler-opt`. It rewrites each block independently (`rewriteBlock`) in a single forward pass, maintaining three maps:

- an *alias* map A sending a temporary index to the value that should replace it downstream;
- an *expression table* E sending a canonical *expression key* to the index of the temporary that currently holds that expression’s value;
- the inverse E^{-1} from a temporary to the key it currently represents.

For each instruction the pass first *substitutes through the alias map*: every operand temporary tm is rewritten to $A(tm)$, chased transitively (`resolveAlias`, with a seen-set to stop on the degenerate cycle). Then, if the instruction is an assignment $td = r$, it (i) *kills* td — dropping td from A , removing any alias pointing at td , and evicting td 's old key from E — and (ii) computes a key for r . The key is defined (`keyOf`) only for *pure* right-hand sides (`IrUsageAnalyzer.isPure`) that are not bare constants, loads, calls, or increments; commutative binary and equality/inequality operators have their operands sorted into a canonical order so that $a + b$ and $b + a$ hash alike. If r 's key is already mapped in E to some *different* temporary te , the assignment is *dropped* and td is aliased to te ($A(td) := te$); otherwise the assignment is kept and E/E^{-1} record that td now holds that key. The terminator is finally rewritten through A as well.

The net effect: the second computation of a pure expression whose operands have not changed since the first is deleted, and all later references to its destination are redirected to the first computation's temporary.

Remark G.76 (Why a reused expression stays available). Reusing te for a later occurrence of expression r is sound only if r would evaluate to the same value at the later point — i.e. if r is *available*: it was computed earlier and none of its operands has been redefined in between. Two facts together guarantee this. First, purity (no load, no call) means r 's value depends only on its operand temporaries, never on memory or control. Second — and this is the load-bearing fact — the pass is single-block and the IR is single-assignment *within* a block (Section G.1), so a temporary that an expression key mentions is never reassigned before the block ends. Availability therefore comes for free from the IR invariant, not from any operand-tracking eviction: while a key sits in E , every temporary it names still holds its original value. The kill step (`killDest`) plays a narrower role. When td is redefined it drops only *its own* forward key — the single key that td currently holds — and removes any alias to td so stale redirections cannot survive; it does *not* scan E for keys that name td as an operand, and under in-block single-assignment it need not, since no such operand is ever reassigned. This is the invariant we verify below.

The availability invariant

Fix a well-formed P and let $P' = \mathcal{P}(P)$. The proof rests on a single invariant tying the pass's tables to runtime values. Consider one block ℓ and a run that enters it in frame φ_0 ; let ρ_t denote the temporary environment after the first t original instructions of ℓ have executed in P .

Lemma G.77 (Tables track values). *At the point where the pass has processed the first t instructions of block ℓ , the following hold for the P -run's environment ρ_t :*

1. if $A(td) = u$ (u a value), then evaluating td and evaluating u under ρ_t give the same word;

2. if $E(\text{key}) = \text{te}$ for a key denoting pure expression r , then te has already been assigned in ℓ , and $\rho_t(\text{te}) = \llbracket r \rrbracket_{\rho_t, \mu}$ — the value r would take if recomputed now.

Proof. Induction on t . At $t = 0$ all three maps are empty and both clauses hold vacuously. Assume both for t and process instruction $I_{t+1} = (\text{td} = r)$ (a store, void call, or non-keyed assignment modifies no map entry whose invariant could break, except the kill of td for any assignment, handled identically).

Kill. Removing td from A and from E/E^{-1} , and deleting aliases pointing at td , can only shrink the maps; the surviving entries still satisfy clauses (1)–(2) because ρ_{t+1} agrees with ρ_t on every temporary other than td , and no surviving key mentions td as an operand whose value just changed — if it did, that key would name a temporary defined *before* this redefinition of td , contradicting in-block single-assignment, since td has now been assigned twice in ℓ . (In well-formed IR this case does not arise at all; the kill is the safe action regardless.)

Reuse branch. Suppose r 's key is already $E(\text{key}) = \text{te} \neq \text{td}$. By clause (2) at t , $\rho_t(\text{te}) = \llbracket r \rrbracket_{\rho_t, \mu}$. The pass drops the assignment and sets $A(\text{td}) := \text{te}$. In the P -run, $\rho_{t+1}(\text{td}) = \llbracket r \rrbracket_{\rho_{t+1}, \mu}$ by the ASSIGN-OP rule. Hence $\rho_{t+1}(\text{td}) = \rho_t(\text{te}) = \rho_{t+1}(\text{te})$ (the step does not touch te), establishing the new clause (1) entry $A(\text{td}) = \text{te}$. The operands of r are unchanged from when te was computed: by in-block single-assignment no operand named by a live key is ever reassigned before the block ends, so a key present in E always denotes a still-current expression — hence clause (2)'s existing entries still hold under ρ_{t+1} .

Record branch. Otherwise the pass keeps the (alias-substituted) assignment and sets $E(\text{key}) := \text{td}$, $E^{-1}(\text{td}) := \text{key}$. The substituted right-hand side $\hat{r}$ satisfies $\llbracket \hat{r} \rrbracket_{\rho_{t+1}} = \llbracket r \rrbracket_{\rho_t}$ by clause (1) at t applied to each rewritten operand (each operand tm was replaced by $A(\text{tm})$, equal in value), and the ASSIGN-OP step gives $\rho_{t+1}(\text{td}) = \llbracket \hat{r} \rrbracket_{\rho_{t+1}}$. Thus $\rho_{t+1}(\text{td}) = \llbracket r \rrbracket_{\rho_{t+1}}$, which is clause (2) for the new entry. $\square$

The simulation relation

For a P -state $\sigma = \langle \pi_\sigma, (\rho, \beta), \mu, \kappa \rangle$ and a P' -state $\sigma' = \langle \pi_{\sigma'}, (\rho', \beta), \mu', \kappa' \rangle$, define $\sigma \approx \sigma'$ when $\mu = \mu'$, $\kappa = \kappa'$, the program points correspond under the deletion map (the target block omits the dropped assignments, so the source index t maps to the target index equal to t minus the number of assignments dropped before it), and for every temporary tm that is *live* at the current point — read by some instruction or terminator not yet executed — we have $\rho(\text{tm}) = \rho'(A^*(\text{tm}))$, where A^* is the transitive alias map the pass would resolve tm through at this point. Equivalently: the target environment holds, under each surviving temporary, exactly the value the source environment holds under that temporary or its alias.

The anti-stuttering measure (Theorem G.9) is the count of consecutive dropped source assignments still unmatched, bounded by the block length; each dropped assignment is

matched by *zero* target steps and strictly decreases it. Initial states are related: empty maps, identical empty ρ , fresh shared μ , empty κ .

Discharging the simulation clauses

Proposition G.78 ($\approx$ is a simulation). *The relation $\approx$ satisfies the three clauses of Theorem G.7.*

Proof. Clause (1) is the initial-state observation. For clause (2) take $\sigma \approx \sigma'$ and $\sigma \rightsquigarrow \tau$, by cases on the source instruction.

A dropped assignment $\text{td} = r$ (reuse branch). The source takes one ASSIGN-OP step setting $\rho(\text{td}) := \llbracket r \rrbracket_{\rho, \mu}$. The target has no corresponding instruction; it stutters (zero steps). We must check $\tau \approx \sigma'$. By Theorem G.77 (2) the target temporary $\text{te} = A(\text{td})$ already holds $\llbracket r \rrbracket_{\rho, \mu}$ in ρ' , and the relation, for any later use of td , consults $A^*(\text{td}) = \text{te}$; thus the value the source just wrote to td matches the value the target reads through the alias. Every other live temporary is unaffected. So $\tau \approx \sigma'$, and the measure strictly decreases.

A kept assignment $\text{td} = r$ (record branch). The source steps on r ; the target steps on the alias-substituted $\hat{r}$. By Theorem G.77 and the relation, the operands of $\hat{r}$ evaluate in ρ' to the same words the operands of r evaluate to in ρ , so both steps write the same word to td and leave μ, κ untouched (pure RHS) — or, for an impure but unkeyed RHS such as *load* or *call*, the same memory/stack effect, since alias substitution only renames equal-valued operands and the framework's STORE/CALL rules depend on operand *values*, which are preserved. Lock-step; $\tau \approx \sigma'$.

A store, void call, or terminator. Operands are alias-substituted to equal-valued ones, so the STORE, CALL, JMP, BR, or RET rule fires with identical effect on both sides — same address and stored value, same callee and arguments, same branch outcome, same returned word. Lock-step.

Clause (3): a terminal source state returns at top level or traps; the matching target terminator was alias-substituted to read an equal-valued operand, so it returns the same word or traps identically. $\square$

Well-formedness and conclusion

The obligations of Theorem G.11 hold. *Control flow*: blocks, labels, and terminators are preserved (only operands inside the terminator are renamed); no edge is added or removed. *Definition dominance*: the reuse branch fires only when te already holds the key, i.e. te was assigned earlier in the same block and hence dominates every later use that the alias redirects to it; deleting td 's assignment removes the only definition of td , but every subsequent reference to td is rewritten to te through A (the terminator and all later instructions pass through *resolveAlias*), so no use of an undefined temporary remains. *Type agreement*: alias substitution preserves operand types — the alias target carries the destination type

recorded when the alias is created (`new IrTemp(existing, assign.dest().type())`), matching the type the substituted-out temporary had — and no operator, store, or cast is otherwise rewritten. Hence $\vdash P' \text{ ok}$.

Theorem G.79 (Semantic preservation of common-subexpression elimination). *For every well-formed program P , $\text{obs}(\mathcal{P}(P)) = \text{obs}(P)$, where $\mathcal{P}$ is `CommonSubexpressionElimination Pass`.*

Proof. If the pass reports no change, immediate. Otherwise [Theorem G.78](#) exhibits a simulation with a well-founded anti-stuttering measure, and well-formedness holds by the paragraph above. [Theorem G.10](#) with [Theorem G.8](#) yields $\text{obs}(P') = \text{obs}(P)$, and [Theorem G.13](#) lifts it across functions and optimizer sweeps. $\square$

G.17 Tiny-function inlining

This section discharges [Theorem G.10](#) for FRISCCc’s tiny-function inliner against the framework of [Section G.1](#). Inlining is the one pass whose rewrite spans the `CALL/RET` rules of [Figure G.1](#): it deletes a call and splices a renamed copy of the callee’s body in its place, so the simulation matches a *multi-step* source trajectory (call, callee body, return) against a multi-step target trajectory (the inlined instructions). The classical treatment is in Muchnick [62, §15.2] and Appel [7, §15.4]; the correctness crux, as always, is that the inlined block reproduces exactly the word the call would have returned, with renamed temporaries avoiding capture.

What the pass does

The implementation is `TinyFunctionInliningPass` in `compiler-opt`. It first collects *inline candidates* (`candidateOf`): a callee g qualifies only if it is extremely restricted —

- its return type is `int32` and it has exactly one basic block;
- it declares *no* local or spill slots and zero local frame bytes (`localsBytes() == 0`, no `LOCAL/SPILL` slot);
- its single block ends in `ret v` with v present, and has at most eight instructions, every one of which is an assignment whose right-hand side is *supported*: an `addr_of_symbol` naming one of g ’s own parameters, an `int32` load, an `int32` binary/comparison/unary/cast operator, or an `int32`-representable constant (`isSupported`).

These constraints make g a *pure leaf* function: with no locals and no spills its only state is its parameter slots and its temporaries, it calls nothing, and its sole effect is to compute and return one `int32` word from its arguments.

At each call site $td = \text{call } g(\bar{a})$ whose callee is such a candidate with matching arity, the pass replaces the single call instruction with a renamed copy of g 's body (`inlineCall`). The splice works symbolically. It threads a map from each of g 's parameter names to the actual argument value a_i , and two working maps: one from a callee temporary that held `addr_of_symbol` p to the parameter name p , and one (`valueTemps`) from a callee temporary to the *value* it denotes in the caller. As it walks g 's assignments:

- an `addr_of_symbol` of a parameter emits nothing and just records the binding $t \mapsto p$;
- a load through such an address-temp resolves to the actual argument a_i for parameter p — modelling that the callee's "read parameter p " becomes "use the caller's argument" directly, since the parameter slot would have been initialized to a_i by `bindParameters`;
- a constant becomes that `int32` constant;
- a pure binary/comparison/unary/cast right-hand side has its operands resolved through `valueTemps` and is re-emitted into a *fresh* caller temporary (index $\geq \text{maxTemp} + 1$), unless it is the temporary g returns, in which case it is emitted directly into the call's destination td .

Finally the callee's returned value — whether a temporary now mapped in `valueTemps` or a constant — is copied into td (`copyAssign` emits either a constant assignment or the canonical copy $td = \text{add } src, \#0$). If any step is unsupported the splice fails and the original call is kept unchanged.

Remark G.80 (The callee touches no memory and no frame slot). The candidate filter is what licenses the splice to ignore the callee's frame entirely. With `localsBytes() == 0` and no `LOCAL` or `SPILL` slots, the only slots g owns are its parameters, and the only way the body reads them is the `addr_of_symbol`-then-load idiom the pass recognises and short-circuits to the actual argument. The supported right-hand sides include no `store` and no `call`, so the body performs *no memory write and no nested call*: its `enter`/`return` cycle in the original semantics allocates a frame, binds parameters, computes pure arithmetic, and tears the frame down, leaving μ exactly as it found it. That is precisely why the inlined code — which allocates no frame at all — can reproduce the call's effect.

The callee evaluation lemma

We first pin down what one call to a candidate g does, in the framework semantics.

Lemma G.81 (A candidate call is a pure value computation). *Let g be an inline candidate, called as $td = \text{call } g(\bar{a})$ from caller state $\sigma = \langle \pi, \varphi, \mu, \kappa \rangle$ with $\bar{w} = \llbracket \bar{a} \rrbracket_\varphi$. Then the sub-run that enters g , executes its block, and returns terminates in $\sigma_{\text{ret}} = \langle \pi^+, \varphi[\rho \mapsto \rho[td \mapsto W]], \mu, \kappa \rangle$, where the memory μ is unchanged and $W = \Phi_g(\bar{w})$ is a function of the arguments alone given by interpreting g 's body with its parameters bound to $\bar{w}$.*

Proof. By CALL the machine forms $(\varphi_g, \mu') = \text{enter}(g, \bar{w}, \mu)$, which allocates g 's slots — here only its parameter slots, since g has no local/spill slots (Theorem G.80) — writes $\bar{w}$ into them, and pushes the return context $(\pi^+, \varphi, \text{td})$. The body is a single block of pure assignments: by inspection of the supported forms, each is an ASSIGN-OP step that reads parameter slots (through `addr_of_symbol` + `load`) and earlier callee temporaries, and writes a callee temporary, never executing a STORE or a CALL. Hence every step leaves the memory at μ' , and μ' differs from μ only in the freshly allocated parameter region of g 's frame. The terminator `ret v` then fires RET-CALLER: it pops the context, writes $W = \llbracket v \rrbracket_{\varphi_g}$ into the caller's `td`, and restores the caller frame φ and pointer π^+ . The callee frame — and with it the only part of memory the call touched — is no longer reachable; modelling frame teardown as reclamation, the observable memory is μ again. Because every read resolves to a parameter slot (hence to some w_i) or to an earlier callee temporary, and every operation is a deterministic function (Theorem G.4), W depends only on $\bar{w}$; call it $\Phi_g(\bar{w})$. $\square$

Lemma G.82 (The splice computes Φ_g). *Suppose the splice of g at the site $\text{td} = \text{call } g(\bar{a})$ succeeds, producing the instruction sequence S . Executed from caller frame φ with $\bar{w} = \llbracket \bar{a} \rrbracket_{\varphi}$, the steps of S leave μ and κ unchanged, write $W = \Phi_g(\bar{w})$ into `td`, and modify only `td` and fresh temporaries that occur nowhere else in the caller.*

Proof. Let θ be the substitution that maps each callee temporary to the caller value the splice associates with it: a parameter-address temp to its parameter name (and thence, through a load, to the actual argument a_i , whose value is $w_i = \llbracket a_i \rrbracket_{\varphi}$), a constant temp to its constant, and an arithmetic temp to the fresh caller temporary the splice allocates for it (or to `td` for the returned one). We claim that, after the splice has emitted the instructions corresponding to the first j callee assignments, each fresh caller temporary $\theta(c)$ holds exactly the value $\rho_j^g(c)$ that callee temporary c holds after g has executed its first j assignments under parameters $\bar{w}$.

Induction on j , mirroring `inlineCall`'s walk. An `addr_of_symbol` p emits nothing and binds $c \mapsto p$; correspondingly g 's temp holds the address of slot p , which the splice never materialises but only consults to turn a later load into the argument value — consistent because the body's only use of that address is a load. A load through such an address resolves to the actual argument w_i , which is exactly $\rho^g(c)$ for the loaded callee temp, since `bindParameters` set slot p to w_i (Theorem G.81). A constant resolves identically on both sides. A pure operator has its operands resolved through the induction hypothesis to the same caller values g 's operands hold, and is re-emitted with the same opcode and result type into a fresh temp; the ASSIGN-OP step therefore writes the same word g writes to its temp. No emitted instruction is a store or call (the supported forms exclude them and `copyAssign` emits only a constant or an `add src, #0`), so μ and κ are untouched throughout. At the end, the returned callee value — temp or constant — equals $\Phi_g(\bar{w})$ by Theorem G.81, and `copyAssign` writes it into `td` (either directly as a constant, or via `td = src + 0`, which evaluates to `src`). The fresh temporaries are drawn above `maxTemp(function)` and are mutually distinct, so they shadow nothing in the caller and capture no caller temporary. Hence S writes $W = \Phi_g(\bar{w})$ to `td`, touches only `td` and those fresh temps, and leaves μ, κ fixed. $\square$

The simulation relation

Fix a well-formed P and let $P' = \mathcal{P}(P)$. If no call was inlined, $P' = P$. Otherwise certain call instructions of P are replaced by their splices S in P' .

For a P -state $\sigma = \langle \pi_\sigma, \varphi, \mu, \kappa \rangle$ and a P' -state $\sigma' = \langle \pi_{\sigma'}, \varphi', \mu', \kappa' \rangle$, define $\sigma \approx \sigma'$ when one of:

Aligned. Neither state is currently *inside* an inlined region. Then the program points correspond under the instruction map that replaces each inlined call by its splice S , the call stacks have the same depth and corresponding return contexts, $\mu = \mu'$, and the frames agree on every temporary live at the current point (the source and target callers share the same temporary names except for the fresh splice temps, which are dead outside their region).

Mid-splice. The source is somewhere in the callee sub-run of an inlined call (it has pushed g 's frame and executed j of g 's assignments), while the target has executed the corresponding j' emitted instructions of S . Then $\mu = \mu'$ on the caller-observable portion (the callee's transient parameter frame on the source side is unobservable, [Theorem G.81](#)), κ' equals κ with the callee context popped, and the caller temporaries agree while the correspondence of [Theorem G.82](#) relates each live callee temp to its fresh caller temp.

The anti-stuttering measure ([Theorem G.9](#)) is the number of source steps inside the current callee sub-run not yet matched by a target step, bounded by g 's block length plus its call/return overhead; it strictly decreases whenever the source advances inside the callee and the target stutters. Initial states are aligned and related: nothing at `main`'s entry is mid-splice, μ and κ start identical, and the temporary environments are both empty.

Discharging the simulation clauses

Proposition G.83 ($\approx$ is a simulation). *The relation $\approx$ satisfies the three clauses of [Theorem G.7](#).*

Proof. Clause (1) holds as noted. For clause (2) take $\sigma \approx \sigma'$ and $\sigma \rightsquigarrow \tau$.

Aligned, non-call step. The current instruction or terminator is identical in P and P' (only inlined calls differ). By determinism the same rule fires; operands agree by the aligned relation, so μ , κ , and the live temporaries evolve identically. Lock-step; $\tau \approx \tau'$, still aligned.

Aligned, an inlined call $\text{td} = \text{call } g(\bar{a})$. The source takes the CALL step, entering g — moving into the mid-splice region with $j = 0$ callee assignments done. The target, whose program point holds the first instruction of the splice S , takes *zero* steps here; we record the transition into the mid-splice case. The relation holds at $j = 0$: μ is unchanged (the callee frame is unobservable), the caller temps are untouched, and no callee temp is yet live. The measure is set to the callee sub-run length and will be drawn down.

Mid-splice, a callee body step. The source executes g 's next assignment. By the correspondence of [Theorem G.82](#), the matching target action is to execute the splice instruction(s) emitted for that assignment (zero target steps for an `addr_of_symbol` or a load/constant the splice folds away; one target step for an emitted operator). When the target stutters, the measure strictly decreases; when it steps, the related fresh caller temp receives the same word the callee temp receives ([Theorem G.82](#)), and μ, κ stay fixed. Either way $\tau \approx \tau'$, still mid-splice.

Mid-splice, the callee's ret v . The source takes the RET-CALLER step: it pops g 's context, writes $W = \Phi_g(\bar{w})$ into the caller's `td`, restores the caller frame, and lands at the return point π^+ — back in the aligned region. The matching target action is the splice's final `copyAssign`, which writes the same W into `td` ([Theorem G.82](#)) and leaves the target at the instruction following S , the point corresponding to π^+ . After this, $\mu = \mu'$ (the callee frame is gone on the source side; the target never had one), $\kappa = \kappa'$ (both at the original depth), and the caller temps agree on `td` and everything else live, with the fresh splice temps now dead. Hence $\tau \approx \tau'$, aligned. The measure reached 0.

Clause (3): a terminal source state is a top-level `ret` or `trap`, necessarily in the aligned region (a candidate callee performs no trapping `addr_index` and never returns at top level, being called, so mid-splice states are non-terminal). In the aligned case the target holds the identical terminator over a frame agreeing on the returned operand, so it reaches a terminal with the same observation. $\square$

Well-formedness and conclusion

The obligations of [Theorem G.11](#) hold. *Control flow:* the splice introduces no block and no terminator — it replaces one assignment-instruction (the call) by a straight-line run of assignment-instructions within the same block (`rewrittenInstructions.addAll(expansion.instructions())`), and the block keeps its label and terminator; every edge is unchanged, and the inlined callee, being a single block ending in `ret`, contributes no new control flow. *Definition dominance:* each emitted instruction defines either a fresh temporary (used only by later emitted instructions in the same straight-line splice, hence dominated) or `td` (defined exactly where the call defined it, so all downstream uses of `td` remain dominated); the address-temp foldings emit nothing and define nothing, removing no needed definition. *Type agreement:* every emitted right-hand side carries the same result type the callee instruction had — `int32` for an arithmetic, cast, load, or constant right-hand side (constants are coerced to `int32` by `asInt32Const`), and `bool` for an intermediate comparison, whose `CmpOp` result type is fixed. Each fresh caller temporary is typed by `assign.dest().type()`, so it inherits exactly the callee temp's type; only the call's destination `td` must match the call's `int32` type, and it does — the candidate filter requires an `int32` return type, so the returned value is `int32` and `copyAssign` writes it into `td` as an `int32` constant or an `int32` add. A `bool`-typed comparison can therefore only ever be an intermediate temp; it is never the returned value bound to `td`. Hence $\vdash P' \text{ ok}$. The original callee functions remain in the program unchanged; inlining a copy does not remove them, so no dangling reference arises even at sites the pass declined to inline.

Theorem G.84 (Semantic preservation of tiny-function inlining). *For every well-formed program P , $\text{obs}(\mathcal{P}(P)) = \text{obs}(P)$, where $\mathcal{P}$ is `TinyFunctionInliningPass`.*

Proof. If no call is inlined, immediate. Otherwise Theorem G.83 exhibits a simulation with a well-founded anti-stuttering measure (the callee sub-run length), and well-formedness holds by the paragraph above. Theorem G.10 and Theorem G.8 give $\text{obs}(P') = \text{obs}(P)$, and Theorem G.13 extends it across the several sites the pass may inline in one sweep and across iterations of the optimizer. $\square$

Remark G.85 (The fresh-temporary discipline is what prevents capture). The single subtle hazard in inlining is variable capture: a callee temporary reusing an index that the caller also uses, so that splicing the body silently overwrites a live caller value. The pass forecloses this by drawing every introduced temporary from indices strictly above the caller’s `maxTemp`, allocated by a counter threaded through the whole function rewrite (`nextTemp`). Because each fresh index is used once, as a callee-temp stand-in, and is dead the moment the splice ends, the mid-splice clause of $\approx$ never has to reconcile a fresh temp with a caller temp of the same name — there are none. This is the implementation-level realisation of the renaming the classical correctness argument performs symbolically.

G.18 Pass: value-range simplification

What the pass does

Value-range simplification is the second abstract interpretation in this appendix, and it differs from global value propagation (Section G.9) in the *shape* of its abstract domain: instead of tracking the exact constant a slot holds, it tracks an integer *interval* $[lo, hi]$ that the slot’s value is guaranteed to lie inside. Intervals are the prototypical non-trivial abstract domain of Cousot and Cousot [20], and they buy something constants cannot: the pass can decide a comparison or a branch whose *exact* value is unknown but whose *outcome* the interval pins down. If analysis proves $i \in [0, 9]$ and the program tests $i < \#10$, the comparison is statically true even though i itself is not a constant; the pass folds the comparison to `#true` and collapses the dependent `br` to an unconditional `jmp`. The implementation is `ValueRangeSimplificationPass` in `compiler-opt`.

The analysis tracks the `int32` local and parameter slots (`trackedInt32Slots`) by an `IntRange` per slot, plus per-temporary ranges and a per-temporary boolean map for comparison results. The block transfer (`transfer`) propagates ranges through loads, stores, negation, and addition/subtraction by a constant, folds exactly-known operands through full arithmetic, and computes comparison outcomes from interval positions (`evaluateComparison`). At joins, incoming ranges are merged by interval *hull* (`IntRange.hull`, the convex union) over the slots all predecessors agree to track (`mergePredecessors`). The analysis iterates to a fixed point, and the rewrite pass performs three simplifications: a comparison with a statically determined outcome becomes the boolean constant `#true` or `#false` (`evaluateComparison` $\rightarrow$

ConstRhs); a conditional branch whose condition temporary is known boolean becomes an unconditional jmp to the taken label ($\text{branchCondition} \rightarrow \text{IrJmpTerm}$); and an `addr_index` with an exactly-known index has that index replaced by its constant. For example, in a loop with a constant trip count whose header has established $i \in [0, 9]$, the test $i < \#10$ folds to true and its guarding `br` becomes a straight jmp into the body. (A symbolic bound such as $[0, n - 1]$ is out of reach: `IntRange` stores concrete `int32` endpoints, and the pass implements no widening, so it cannot pin a counter to a symbolic upper bound.)

The claim

Theorem G.86 (Value-range simplification preserves semantics). *Let $\mathcal{P}_{\text{vrs}}$ be the value-range-simplification pass. For every well-formed program P (Theorem G.1), $\text{obs}(\mathcal{P}_{\text{vrs}}(P)) = \text{obs}(P)$.*

As with global value propagation, the pass adds, deletes, and reorders no blocks, and the one structural change it makes — turning a `br` into a `jmp` — is sound *precisely because* the analysis has proven the dropped edge unreachable from this point. The argument again factors into soundness of the abstraction and a value/direction preservation of the rewrite.

The abstraction and its soundness

Definition G.87 (Concretization of a range fact). A *range fact* $\mathcal{R}$ is a finite partial map from tracked `int32` slot names to intervals $[lo, hi]$ with $lo \leq hi$. A state $\sigma = \langle (f, \ell, i), (\rho, \beta), \mu, \kappa \rangle$ is *consistent with* $\mathcal{R}$, written $\sigma \models \mathcal{R}$, when for every slot $s \in \text{dom } \mathcal{R}$,

$$lo \leq \text{load}_{\text{int32}}(\mu, \text{addr}_{\varphi}(s)) \leq hi, \quad \text{where } \mathcal{R}(s) = [lo, hi].$$

The temporary companion is identical with $\llbracket \text{tn} \rrbracket_{\varphi}$ in place of the slot load. $\gamma(\mathcal{R}) = \{\sigma : \sigma \models \mathcal{R}\}$ is the concretization. The order is interval inclusion lifted pointwise (and reverse map-inclusion on domains): $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ when $\mathcal{R}_2$'s domain is a subset of $\mathcal{R}_1$'s and each shared interval of $\mathcal{R}_1$ is contained in $\mathcal{R}_2$'s. The hull merge computes the join in this order, and γ is monotone: a wider interval, or a smaller domain, describes more states.

Lemma G.88 (Range transfer soundness). *Let $\mathcal{R}$ hold at a block B 's entry and let $\mathcal{R}'$ be the fact transfer produces at B 's exit. If $\sigma \models \mathcal{R}$ and the original program steps σ through B to τ at B 's exit, then $\tau \models \mathcal{R}'$, and the same holds at every intermediate point.*

Proof. By induction over B 's instructions, maintaining $\sigma' \models \text{slotRanges}$ together with the soundness of the per-temporary maps (tempRanges : $\llbracket \text{tn} \rrbracket_{\varphi} \in \text{tempRanges}[\text{tn}]$; boolTemps : $\llbracket \text{tn} \rrbracket_{\varphi} = 1$ iff the boolean flag is true). The cases mirror transfer.

Constants and exact arithmetic. A constant `int32` right-hand side records the singleton $[k, k]$, which contains k ; a boolean constant records its flag. An integer binop both of

whose operands are exact singletons folds to the singleton of the integer the ASSIGN-OP rule computes (evaluateBinOpRange’s exact branch, using the same operations as Figure G.1 under Theorem G.2, and declining div/mod by zero). Negation of an exact operand records the negated singleton.

Interval arithmetic. For $x + c$, $c + x$, or $x - c$ with c exact and x ranging over $[lo, hi]$, the transfer records $[lo + c, hi + c]$ (IntRange.add), returning *no fact* (null) if the shift would overflow the int32 range — the conservative escape that keeps the recorded interval a true over-approximation of the wrapping machine value. Since $x \in [lo, hi]$ implies $x + c \in [lo + c, hi + c]$ when no overflow occurs, soundness holds.

Load and store of a tracked slot. A load of a tracked int32 slot copies the slot’s interval to the destination temporary; by $\sigma' \models \text{slotRanges}$ the loaded word lies in that interval, preserving temporary soundness. A store of a range-known value to a tracked slot records that range on the slot, which the STORE rule then makes true of μ ; a store of an unknown value removes the slot’s binding (sound: no claim is made).

Comparisons. evaluateComparison returns a definite boolean *only* when the operand intervals are separated or coincident enough to force the outcome — e.g. *LT* yields true when $hi_L < lo_R$ and false when $lo_L \geq hi_R$. Theorem G.89 below verifies each clause. When it returns a boolean, the recorded boolTemps flag is the value the original cmp computes on σ , so boolean soundness holds; when it returns nothing, no flag is recorded.

Conservative clears. A call or increment/decrement clears slotRanges; a store through an address that does not resolve to a tracked slot clears it as well. Clearing empties the domain, restoring $\models$ vacuously.

Each step preserves the conjoined invariant, so the exit fact $\mathcal{R}'$ satisfies $\tau \models \mathcal{R}'$. $\square$

Lemma G.89 (Comparison decisions are correct). *Suppose $\llbracket e_L \rrbracket_\varphi \in [lo_L, hi_L]$ and $\llbracket e_R \rrbracket_\varphi \in [lo_R, hi_R]$. Then each boolean that evaluateComparison returns equals the value of the corresponding int32 comparison on $\llbracket e_L \rrbracket_\varphi$ and $\llbracket e_R \rrbracket_\varphi$.*

Proof. We verify the order cases; equality and inequality are symmetric. Write $L = \llbracket e_L \rrbracket_\varphi$, $R = \llbracket e_R \rrbracket_\varphi$.

For *LT*, the code returns true when $hi_L < lo_R$; then $L \leq hi_L < lo_R \leq R$, so $L < R$ holds. It returns false when $lo_L \geq hi_R$; then $L \geq lo_L \geq hi_R \geq R$, so $L < R$ is false. For *LE*, true requires $hi_L \leq lo_R$, giving $L \leq hi_L \leq lo_R \leq R$; false requires $lo_L > hi_R$, giving $L > R$, so $L \leq R$ is false. *GT* and *GE* are the mirror images and check out by the same inequalities. For *EQ*, the code returns false when the intervals are disjoint ($hi_L < lo_R$ or $hi_R < lo_L$), where no common value exists, and true only when both operands are the *same* exact singleton, where $L = R$ necessarily; *NE* is its negation. In every branch where a boolean is returned, it equals the true comparison outcome on L and R ; in the remaining (overlapping) configurations the code returns nothing and no decision is made. $\square$

Lemma G.90 (Analysis soundness at every reachable point). *After the worklist fixed point, the entry fact $\text{inStates}[B]$ of every block B is consistent with every state reaching B 's entry along any run of P from $\text{init}(P)$.*

Proof. Identical in structure to [Theorem G.44](#): the entry block carries $\top$ (empty map), consistent with all states; every other block's entry fact is the hull-merge of its predecessors' transferred exit facts. Induction on run length, using [Theorem G.88](#) within a block and, at a control-flow edge, the fact that each predecessor's contribution is $\sqsubseteq$ the hull-merge so that by monotonicity of γ the reaching state stays consistent. The hull is a sound join because $a \in I$ implies $a \in \text{hull}(I, J)$ for any J (the convex union only widens each interval), so a state consistent with one predecessor's exit fact is consistent with the merged entry fact. Calls clear slot ranges, so no range claim survives a callee's writes. $\square$

The simulation relation and the rewrite

Definition G.91 (VRS simulation). Relate σ of P to σ' of P' by state identity, exactly as in [Theorem G.45](#), with anti-stuttering measure $\mathbf{m} \equiv 0$. The rewrite preserves all of frame, memory, call stack, and the program point (a folded comparison and a $\text{br} \rightarrow \text{jmp}$ both leave the next control point unchanged), so identity is again the right candidate.

Lemma G.92 (Rewrite preserves values and branch directions). *Let an instruction or terminator at point p be rewritten using the fixed-point fact $\mathcal{R} = \text{fact}(p)$, and let σ reach p . Then (i) a folded comparison evaluates to the same boolean as the original; (ii) an `addr_index` with a constant-substituted index computes the same address; and (iii) a `br` rewritten to a `jmp` ℓ transfers control to the same successor block the `br` would have taken in σ .*

Proof. By [Theorem G.90](#), $\sigma \models \mathcal{R}$, and the local maps rebuilt by the rewrite pass over the block prefix are sound for σ .

(i) A comparison is folded to a boolean constant only when `evaluateComparison` returns a definite outcome on the operand ranges that are sound for σ ; by [Theorem G.89](#) that outcome is the value the original comparison computes on σ 's operands, so the constant right-hand side evaluates identically.

(ii) The index of an `addr_index` is replaced by a constant only when its range is an exact singleton $[k, k]$, in which case $\llbracket \text{index} \rrbracket_{\varphi} = k$ by temporary soundness, so the address arithmetic of [Figure G.1](#) yields the same address; since the original index was already in bounds or already trapping, the bounds check on the constant index takes the identical branch (in-bounds stays in-bounds; an out-of-bounds constant traps exactly as the original out-of-bounds value did).

(iii) A `br` c, ℓ_t, ℓ_f is rewritten to `jmp` ℓ_t (resp. ℓ_f) only when the condition temporary c carries a boolean flag true (resp. false) in `boolTemps`; by boolean soundness $\llbracket c \rrbracket_{\varphi} \neq 0$ (resp. $= 0$), so the BR-T rule (resp. BR-F) on the original would branch to ℓ_t (resp. ℓ_f) — exactly

the unconditional target. The `jmp` thus reaches the same block in one step, and the *other*, untaken edge is dropped without changing where σ goes. $\square$

Discharging the obligations

Proof of Theorem G.86. Well-formedness. The pass rewrites no block list and introduces no new label: a folded comparison stays an assignment to the same temporary; a `br` $\rightarrow$ `jmp` replaces a two-target terminator by a one-target terminator whose target is one of the original two labels, so it still names a block of the same function and the block still ends in exactly one terminator — control-flow integrity holds. No definition is deleted (a folded comparison still assigns its destination), so definition dominance holds; the substituted index constant and the boolean constant carry the consumer’s type by construction (`IrPrimitiveType.INT32`, `IrPrimitiveType.BOOL`), so type agreement holds. Thus $\vdash \mathcal{P}_{\text{vrs}}(P)$ *ok* by Theorem G.11.

Simulation. Take $\approx$ to be state identity (Theorem G.91). Clause (1) is immediate: the pass changes no slot table, globals, or entry pointer, so $\text{init}(P) = \text{init}(P')$. Clause (2): let $\sigma \approx \sigma'$, i.e. $\sigma = \sigma'$, and $\sigma \rightsquigarrow \tau$. If the current instruction was untouched, the same rule fires on both sides to an identical τ . If it was a folded comparison, the rewritten constant right-hand side evaluates to the same boolean (Theorem G.92(i)), so `ASSIGN-OP` binds the same word and $\tau' = \tau$. If it was an `addr_index` with a substituted constant index, the same address is computed (Theorem G.92(ii)) and the trap (if any) fires identically. If the terminator was a `br` folded to a `jmp`, then by Theorem G.92(iii) the source `br` branches to the same block the target `jmp` reaches, so the single source step and single target step land in the identical state $\tau' = \tau$. In every case $\tau \approx \tau'$ in true lock-step. Clause (3): a terminal `ret v` matches the identical target state (the pass never rewrites a return value), observing the same word; a bounds trap matches a bounds trap by Theorem G.92(ii).

All clauses hold, so by Theorem G.8 $\text{obs}(\mathcal{P}_{\text{vrs}}(P)) = \text{obs}(P)$, and Theorem G.12 carries termination and trapping behaviour. $\square$

Implementation pointers

`ValueRangeSimplificationPass.simplifyFunction` runs the worklist fixed point and then rewrites; `IntRange` is the interval domain with `hull` as `join` and `add` carrying the overflow guard; `transfer` is the transfer function; `evaluateComparison` is the comparison-decision oracle proved correct in Theorem G.89; `branchCondition` drives the `br` $\rightarrow$ `jmp` rewrite. Three honest notes on the gap between code and theorem. First, the domain is intervals *without* widening, and `IntRange.add` bails to `null` on any arithmetic that would leave the `int32` range; the analysis therefore loses precision on loops whose induction variable could overflow, but never loses *soundness*, since dropping a slot’s interval only enlarges $\gamma(\mathcal{R})$. Second, the pass only *rewrites the terminator* when a branch is proven one-directional; it does not delete the now-unreachable successor block — that cleanup is left to `UnreachableBlockEliminationPass`, whose own proof discharges the control-flow obligation for the orphaned block. Third, exact-singleton folding overlaps in effect with constant folding and global value propagation;

the passes are independently sound, and [Theorem G.13](#) guarantees that running them together, in any order and to a fixed point, preserves semantics.

Bibliography

- [1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. “Code Generation Using Tree Matching and Dynamic Programming”. In: *ACM Transactions on Programming Languages and Systems* 11.4 (1989), pp. 491–516. DOI: [10.1145/69558.75700](https://doi.org/10.1145/69558.75700).
- [2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd. Boston, MA: Addison-Wesley, 2006.
- [3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. The original “red dragon” edition; predecessor to [2]. Addison-Wesley, 1986.
- [4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2001.
- [5] G. M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference (AFIPS ’67)*. ACM, 1967, pp. 483–485. DOI: [10.1145/1465482.1465560](https://doi.org/10.1145/1465482.1465560).
- [6] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge, UK: Cambridge University Press, 1998.
- [7] A. W. Appel. *Modern Compiler Implementation in Java*. 2nd. Cambridge, UK: Cambridge University Press, 2002.
- [8] J. Aycock. “A Brief History of Just-in-Time”. In: *ACM Computing Surveys* 35.2 (2003), pp. 97–113. DOI: [10.1145/857076.857077](https://doi.org/10.1145/857076.857077).
- [9] D. Basch and M. Kovač. *Osnove procesora FRISC*. Croatian. Ed. by D. Antičić. 2nd ed. Primary reference for the FRISC architecture and instruction set; 2nd edition, edited by Davor Antičić. Zagreb: Antičić, 2004. 248 pp.
- [10] J. R. Bell. “Threaded Code”. In: *Communications of the ACM* 16.6 (1973), pp. 370–372. DOI: [10.1145/362248.362270](https://doi.org/10.1145/362248.362270).
- [11] G. Bierman and B. Goetz. *JEP 441: Pattern Matching for switch (Final)*. OpenJDK Java Enhancement Proposal. Delivered in Java 21 LTS; standardises sealed-type pattern matching and exhaustiveness checking. 2023.
- [12] N. Binkert, B. Beckmann, G. Black, et al. “The gem5 Simulator”. In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7. DOI: [10.1145/2024716.2024718](https://doi.org/10.1145/2024716.2024718).
- [13] P. Briggs, K. D. Cooper, and L. Torczon. “Improvements to Graph Coloring Register Allocation”. In: *ACM Transactions on Programming Languages and Systems* 16.3 (1994), pp. 428–455. DOI: [10.1145/177492.177575](https://doi.org/10.1145/177492.177575).

- [14] J. A. Brzozowski. “Derivatives of Regular Expressions”. In: *Journal of the ACM* 11.4 (1964), pp. 481–494. DOI: [10.1145/321239.321249](https://doi.org/10.1145/321239.321249).
- [15] G. J. Chaitin. “Register Allocation & Spilling via Graph Coloring”. In: *SIGPLAN Notices* 17.6 (1982), pp. 98–105. DOI: [10.1145/872726.806984](https://doi.org/10.1145/872726.806984).
- [16] J. Cocke and J. T. Schwartz. *Programming Languages and their Compilers: Preliminary Notes*. Second revised. Early systematic treatment of strength reduction and induction-variable analysis. Courant Institute of Mathematical Sciences, New York University, 1970.
- [17] K. D. Cooper, M. W. Hall, and L. Torczon. “An experiment with inline substitution”. In: *Software: Practice and Experience* 21.6 (1991). Empirical study of when inlining helps and when it harms; basis for size-bounded heuristics., pp. 581–601. DOI: [10.1002/spe.4380210604](https://doi.org/10.1002/spe.4380210604).
- [18] K. D. Cooper and L. Torczon. *Engineering a Compiler*. 2nd. Burlington, MA: Morgan Kaufmann, 2011.
- [19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 3rd. MIT Press, 2009.
- [20] P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 1977, pp. 238–252. DOI: [10.1145/512950.512973](https://doi.org/10.1145/512950.512973).
- [21] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Transactions on Programming Languages and Systems* 13.4 (1991), pp. 451–490. DOI: [10.1145/115372.115320](https://doi.org/10.1145/115372.115320).
- [22] J. W. Davidson and C. W. Fraser. “The Design and Application of a Retargetable Peephole Optimizer”. In: *ACM Transactions on Programming Languages and Systems* 2.2 (1980). The retargetable peephole optimiser whose ideas the code generator’s local clean-ups echo., pp. 191–202. DOI: [10.1145/357094.357098](https://doi.org/10.1145/357094.357098).
- [23] F. DeRemer and T. Pennello. “Efficient Computation of LALR(1) Look-Ahead Sets”. In: *ACM Transactions on Programming Languages and Systems* 4.4 (1982), pp. 615–649. DOI: [10.1145/69622.357187](https://doi.org/10.1145/69622.357187).
- [24] F. L. DeRemer. “Practical Translators for LR(k) Languages”. Ph.D. thesis. Cambridge, MA: Massachusetts Institute of Technology, 1969.
- [25] J. Earley. “An Efficient Context-Free Parsing Algorithm”. In: *Communications of the ACM* 13.2 (1970). The general context-free parser that handles any grammar in cubic time; the algorithm named, but previously uncited, in the parsing chapter., pp. 94–102. DOI: [10.1145/362007.362035](https://doi.org/10.1145/362007.362035).
- [26] M. A. Ertl and D. Gregg. “The Structure and Performance of Efficient Interpreters”. In: *Journal of Instruction-Level Parallelism* 5 (2003), pp. 1–25.
- [27] FER, University of Zagreb. *Prevođenje programskih jezika (Programming Language Translation) – course materials*. <https://www.fer.unizg.hr/predmet/ppj>. 2024.

- [28] P. J. Fleming and J. J. Wallace. “How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results”. In: *Communications of the ACM* 29.3 (1986), pp. 218–221. DOI: [10.1145/5666.5673](https://doi.org/10.1145/5666.5673).
- [29] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. “Engineering a Simple, Efficient Code-Generator Generator”. In: *ACM Letters on Programming Languages and Systems* 1.3 (1992), pp. 213–226. DOI: [10.1145/151640.151642](https://doi.org/10.1145/151640.151642).
- [30] A. Gal, B. Eich, M. Shaver, et al. “Trace-Based Just-in-Time Type Specialization for Dynamic Languages”. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009, pp. 465–478. DOI: [10.1145/1542476.1542528](https://doi.org/10.1145/1542476.1542528).
- [31] L. George and A. W. Appel. “Iterated Register Coalescing”. In: *ACM Transactions on Programming Languages and Systems* 18.3 (1996), pp. 300–324. DOI: [10.1145/229542.229546](https://doi.org/10.1145/229542.229546).
- [32] D. Goldberg. “What Every Computer Scientist Should Know About Floating-Point Arithmetic”. In: *ACM Computing Surveys* 23.1 (1991), pp. 5–48. DOI: [10.1145/103162.103163](https://doi.org/10.1145/103162.103163).
- [33] D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. 2nd. Springer, 2008. DOI: [10.1007/978-0-387-68954-8](https://doi.org/10.1007/978-0-387-68954-8).
- [34] R. Harper. *Practical Foundations for Programming Languages*. 2nd. Cambridge University Press, 2016.
- [35] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th. Morgan Kaufmann, 2017.
- [36] J. L. Henning. “SPEC CPU2000: Measuring CPU Performance in the New Millennium”. In: *Computer* 33.7 (2000), pp. 28–35. DOI: [10.1109/2.869367](https://doi.org/10.1109/2.869367).
- [37] R. Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (1969). The type-inference result later rediscovered by Milner; together they name the Hindley–Milner system the semantics chapter alludes to., pp. 29–60. DOI: [10.2307/1995158](https://doi.org/10.2307/1995158).
- [38] J. E. Hopcroft. “An $n \log n$ Algorithm for Minimizing States in a Finite Automaton”. In: *Theory of Machines and Computations*. Ed. by Z. Kohavi and A. Paz. New York: Academic Press, 1971, pp. 189–196.
- [39] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd. Pearson, 2006.
- [40] ISO/IEC. *ISO/IEC 9899:2018 – Information technology – Programming languages – C*. International standard. 2018.
- [41] S. C. Johnson. *Yacc: Yet Another Compiler Compiler*. Computing Science Technical Report 32. Murray Hill, NJ: Bell Laboratories, 1975.
- [42] G. Kahn. “Natural Semantics”. In: *STACS 87: 4th Annual Symposium on Theoretical Aspects of Computer Science*. Vol. 247. Lecture Notes in Computer Science. Springer, 1987, pp. 22–39. DOI: [10.1007/BFb0039592](https://doi.org/10.1007/BFb0039592).

- [43] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall, 1988.
- [44] G. A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Conference Record of the First ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 1973, pp. 194–206. DOI: [10.1145/512927.512945](https://doi.org/10.1145/512927.512945).
- [45] S. C. Kleene. “Representation of Events in Nerve Nets and Finite Automata”. In: *Automata Studies* (1956), pp. 3–41. DOI: [10.1515/9781400882618-002](https://doi.org/10.1515/9781400882618-002).
- [46] K. Knežević. *FRISCcc Companion: A Build-Along tinyC Compiler*. Open-source build-along project for this book: per-chapter starter, reference solution, and test modules for a tinyC compiler. 2026.
- [47] K. Knežević. *FRISCcc: A C-Subset Compiler for the FRISC Architecture*. Open-source source code of the FRISCcc compiler dissected throughout this book. 2026.
- [48] D. E. Knuth. “On the Translation of Languages from Left to Right”. In: *Information and Control* 8.6 (1965), pp. 607–639. DOI: [10.1016/S0019-9958\(65\)90426-2](https://doi.org/10.1016/S0019-9958(65)90426-2).
- [49] D. E. Knuth. “Structured Programming with go to Statements”. In: *ACM Computing Surveys* 6.4 (1974). Source of the often-quoted “premature optimization is the root of all evil.”, pp. 261–301. DOI: [10.1145/356635.356640](https://doi.org/10.1145/356635.356640).
- [50] D. E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. 3rd. Addison-Wesley, 1997.
- [51] D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. 3rd. Addison-Wesley, 1998.
- [52] J. R. Larus. *SPIM: A MIPS32 Simulator*. <http://spimsimulator.sourceforge.net/>. Teaching simulator for the MIPS instruction set; see also the MARS simulator. No longer actively maintained as of 2023. 2010.
- [53] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2004, pp. 75–86. DOI: [10.1109/CGO.2004.1281665](https://doi.org/10.1109/CGO.2004.1281665).
- [54] T. Lengauer and R. E. Tarjan. “A Fast Algorithm for Finding Dominators in a Flowgraph”. In: *ACM Transactions on Programming Languages and Systems* 1.1 (1979). The near-linear dominator algorithm presupposed by every SSA-construction method discussed in the closing chapter., pp. 121–141. DOI: [10.1145/357062.357071](https://doi.org/10.1145/357062.357071).
- [55] X. Leroy. “Formal Certification of a Compiler Back-End”. In: *Proceedings of POPL ’06*. 2006, pp. 42–54. DOI: [10.1145/1111037.1111042](https://doi.org/10.1145/1111037.1111042).
- [56] X. Leroy. “Formal Verification of a Realistic Compiler”. In: *Communications of the ACM* 52.7 (2009). Overview of CompCert, a verified C compiler whose semantic-preservation theorems are the closest published cousin to Appendix L., pp. 107–115. DOI: [10.1145/1538788.1538814](https://doi.org/10.1145/1538788.1538814).
- [57] M. E. Lesk and E. Schmidt. *Lex – A Lexical Analyzer Generator*. Computing Science Technical Report 39. Bell Laboratories, 1975.

- [58] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. *System V Application Binary Interface, AMD64 Architecture Processor Supplement*. Technical Report. Draft version 0.99.6. The Santa Cruz Operation / x86-64 psABI maintainers, 2013.
- [59] W. M. McKeeman. “Peephole Optimization”. In: *Communications of the ACM* 8.7 (1965), pp. 443–444. DOI: [10.1145/364995.365000](https://doi.org/10.1145/364995.365000).
- [60] W. M. McKeeman. *Differential Testing for Software*. Digital Technical Journal 1. Digital Equipment Corporation, 1998, pp. 100–107.
- [61] R. McNaughton and H. Yamada. “Regular Expressions and State Graphs for Automata”. In: *IRE Transactions on Electronic Computers* EC-9.1 (1960), pp. 39–47. DOI: [10.1109/TEC.1960.5221603](https://doi.org/10.1109/TEC.1960.5221603).
- [62] S. S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann, 1997.
- [63] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. “Producing Wrong Data Without Doing Anything Obviously Wrong!” In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, 2009, pp. 265–276. DOI: [10.1145/1508244.1508275](https://doi.org/10.1145/1508244.1508275).
- [64] P. Naur, J. W. Backus, et al. “Revised Report on the Algorithmic Language ALGOL 60”. In: *Communications of the ACM* 6.1 (1963), pp. 1–17. DOI: [10.1145/366193.366201](https://doi.org/10.1145/366193.366201).
- [65] J. von Neumann. *First Draft of a Report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania. Reprinted in *IEEE Annals of the History of Computing* 15(4):27–75, 1993. 1945. DOI: [10.1109/85.238389](https://doi.org/10.1109/85.238389).
- [66] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999. DOI: [10.1007/978-3-662-03811-6](https://doi.org/10.1007/978-3-662-03811-6).
- [67] R. Nystrom. *Crafting Interpreters*. Stillwater, OK: Genever Benning, 2021.
- [68] D. A. Patterson. “Reduced Instruction Set Computers”. In: *Communications of the ACM* 28.1 (1985), pp. 8–21. DOI: [10.1145/2465.214917](https://doi.org/10.1145/2465.214917).
- [69] B. C. Pierce. *Types and Programming Languages*. Cambridge, MA: MIT Press, 2002.
- [70] G. D. Plotkin. *A Structural Approach to Operational Semantics*. Tech. rep. DAIMI FN-19. Aarhus University, 1981.
- [71] M. Poletto and V. Sarkar. “Linear Scan Register Allocation”. In: *ACM Transactions on Programming Languages and Systems* 21.5 (1999), pp. 895–913. DOI: [10.1145/330249.330250](https://doi.org/10.1145/330249.330250).
- [72] C. Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 2003.
- [73] M. O. Rabin and D. Scott. “Finite Automata and Their Decision Problems”. In: *IBM Journal of Research and Development* 3.2 (1959), pp. 114–125. DOI: [10.1147/rd.32.0114](https://doi.org/10.1147/rd.32.0114).
- [74] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. “Virtual Machine Showdown: Stack Versus Registers”. In: *ACM Transactions on Architecture and Code Optimization* 4.4 (2008), pp. 1–36. DOI: [10.1145/1328195.1328197](https://doi.org/10.1145/1328195.1328197).
- [75] M. Sipser. *Introduction to the Theory of Computation*. 3rd. Cengage Learning, 2013.

- [76] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.
- [77] S. Srbljić. *Prevodenje programskih jezika*. Croatian. Standard Croatian-language compiler-construction text for the FER *Prevodenje programskih jezika* course; source of the grammar conventions and non-terminal names used throughout (see Appendix A). Zagreb: Element, 2009.
- [78] S. Srbljić. *Uvod u teoriju računarstva*. Croatian. Croatian-language introduction to the theory of computation: regular languages, finite automata, context-free grammars, and the formal results underlying the front end. Zagreb: Element, 2010.
- [79] G. L. Steele. *RABBIT: A Compiler for SCHEME*. AI Memo 474. MIT Artificial Intelligence Laboratory, 1978.
- [80] K. Thompson. “Programming Techniques: Regular Expression Search Algorithm”. In: *Communications of the ACM* 11.6 (1968), pp. 419–422. DOI: [10.1145/363347.363387](https://doi.org/10.1145/363347.363387).
- [81] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic, 1985.
- [82] H. S. Warren. *Hacker’s Delight*. 2nd. Addison-Wesley, 2013.
- [83] M. N. Wegman and F. K. Zadeck. “Constant Propagation with Conditional Branches”. In: *ACM Transactions on Programming Languages and Systems* 13.2 (1991), pp. 181–210. DOI: [10.1145/103135.103136](https://doi.org/10.1145/103135.103136).
- [84] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.
- [85] N. Wirth. *Compiler Construction*. Author-revised PDF edition of the 1996 Addison-Wesley book (ISBN 978-0201403534). Self-published, 2017.
- [86] A. K. Wright and M. Felleisen. “A Syntactic Approach to Type Soundness”. In: *Information and Computation* 115.1 (1994), pp. 38–94. DOI: [10.1006/inco.1994.1093](https://doi.org/10.1006/inco.1994.1093).
- [87] R. Yates. *Fixed-Point Arithmetic: An Introduction*. Technical reference 20090811. Digital Signal Labs, 2009.
- [88] I. Žužak. *FRISCjs: FRISC processor simulator in JavaScript*. <https://github.com/izuzak/FRISCjs>. Source code of the FRISC simulator used throughout this book; live web application at <https://ivanzuzak.info/FRISCjs/webapp/>. 2011.

Index

- ablation, 330
 - dynamic versus static, 333
 - procedure, 330
- abstract interpretation, 177
- abstract syntax tree, 9
- accept state, 56
- ACTION table, 84
- activation record, 217
- addressing mode, 206, 365
- alias analysis, 344
- alphabet, 44
- alternation, 44
- ambiguity, 73
- Amdahl's law, 335
- argument passing, 205
- artifact contract, 24
- assembler, 236
- associativity, 74
- augmentation (grammar), 77

- back end, 12
- back-patching, 280
- backreference, 45
- backtracking, 45
- basic block, 137, 261
- benchmark suite, 326
- benchmarking
 - counted metrics, 326
 - work counts, 299, 307
- bisimulation, 464
- bottom-up parsing, 75
- bounds check, 201, 226, 266, 308
- bounds checking, 226
- Brzowski derivatives, 69
- byte-addressable memory, 259
- bytecode, 276
- bytecode compiler, 279

- call overhead, 302
- call stack, 285
 - explicit, 275
 - host, 265
- calling convention, 190, 216
- canonical collection, 83
- canonical-forms lemma, 126
- carry flag, 225
- case study, 299
- cast
 - explicit, 115
 - implicit, 115
- character class, 44
- Chomsky hierarchy, 74
- closure (LR), 79
- code generation, 15, 25, 189
- comparison materialisation, 199
- compilation, 256
- compilation versus interpretation, 234
- compiler, 3
- compiler architecture, 23
- compiler pipeline, 3
- concatenation, 44
- condition code, 367
- condition flags, 238
- const qualifier, 118
- constant folding, 316
- context-free grammar, 72
- control flow
 - flattening, 280
- control-flow label, 137
- control-flow simplification, 331
- convergence, 335
- CSE, *see* common subexpression
 - elimination
- cycle trace, 244

- dangling else, 90
- data section, 205
- data-flow analysis, 168
- dataflow analysis, 267
- definitely-returns predicate, 118
- determinism
 - LR(1), 87
- DFA, 50
- DFA minimization, 53
- diagnostic, 24
- differential testing, 269, 288
- disassembler, 280
- dispatch, 257
 - bytecode, 282
 - stack versus register, 346
 - switch, 282
 - unit of work, 289
- dispatch loop, 282
- division
 - restoring, 220
 - software, 220
- division by zero, 222
- dominance, 146
- dominator, 350
- dynamic instruction count, 326
- dynamic instruction mix, 245

- Earley parsing, 103
- endianness, 236
- entry stub, 203, 217
- environment (interpreter), 259
- epilogue, 204, 217
- epsilon-closure, 49
- error code, 227
- error recovery, 92
- Euclid's algorithm, 304
- evaluation, 261
 - right-hand side, 264
- expansion factor, 289, 311
 - flat (VM), 316
- explicit cast, 108
- extension, 341

- fail-fast, 65
- faithfulness, 288

- fetch-decode-execute, 235
- Fibonacci, 300
- finite automaton, 7
 - deterministic, 50
 - nondeterministic, 46
- FIRST set, 79, 95
- fixed-point
 - conversions, 224
 - division, 225
 - multiplication, 225
 - Q16.16, 223
- fixed-point arithmetic, 223, 314
- flex, 62
- floating-point
 - IEEE 754, 225
- frame
 - interpreter, 259
 - IR, 135
- frame pointer, 192, 216
- frame slot, 135, 193
- FRISC, 3
 - immediate field, 192
 - load-store architecture, 190
 - missing instructions, 190
- FRISC instruction
 - ADD, 366
 - AND, 366
 - ASHR, 366
 - CALL, 367
 - CMP, 366
 - HALT, 367
 - JP, 367
 - JR, 367
 - LOAD, 366
 - MOVE, 366
 - OR, 366
 - POP, 367
 - PUSH, 367
 - RET, 367
 - SHL, 366
 - SHR, 366
 - STORE, 366
 - SUB, 366
 - XOR, 366
- FRISC instruction set, 361

- front end, 12
 - new, 353
- function signature, 124
- garbage collection, 344
- GCD, 304
- geometric mean, 340
- GLR parsing, 103
- goto (LR), 81
- GOTO table, 84
- grammar, 72
 - FRISCcc, 92
- halt, 241
- halting problem, 242
- helper budget, 305
- helper routine, 190, 215
- helper-cycle budget, 333
 - suite aggregate, 334
- hexadecimal immediate, 223
- Hindley-Milner, 128
- Hopcroft minimization, 53
- Horner's method, 311
- hot path, 352
- identifier, 54
- immediate operand, 192, 364
- implicit conversion, 108
- implicit promotion, 108
- induction variable, 175
- instruction count, 299
- instruction cycle, 235
- instruction decode, 238
- instruction encoding, 238, 278, 364
- instruction format, 364
- instruction selection, 196
- instruction set
 - extending, 352
- intermediate representation, 12, 24, 131
 - as hourglass waist, 343
 - typed, 131
 - unit of reuse, 353
- interpretation-Compilation spectrum, 256
- interpreter, 256
- invariant, 466
 - load-forwarding known value, 514
- IR grammar, 377
 - three-address form, 394
- IR interpreter, 256
- IR temporary, 132
- IR verifier, 148
- IR-to-IR pass, 161
- item set, 79
- just-in-time compilation, 276, 350
- keyword, 54
- kinematics, 314
- Kleene closure, 44
- Kleene star, 44
- knapsack, 308
- label generation, 200
- LALR, 102
- language, 44
- left recursion, 75
- lex, 70
- lexeme, 43
- lexer generator, 70
- lexer mode, 60
- lexer specification, 63, 377
- lexical analysis, 5
- lexical error, 65
- lexical scope, 106
- LICM, *see* loop-invariant code motion
- linkage region, 216
- LL parsing, 75
- load-store architecture, 30
- lookahead, 77
- loop lowering, 143
- loop-depth counter, 117
- loop-invariant code motion, 333
- lowering
 - expressions, 131
 - l-value, 140
 - r-value, 140
 - statements, 131
 - to bytecode, 279
- LR parser, 71
- LR parsing, 75
- LR(0) item, 77
- LR(1), 71

- LR(1) item, 77
- LR(k), 76
- lvalue, 106, 128
- machine cycle, 374
 - per-cycle hook, 374
- machine state, 234
- macro (lexer), 63
- maximal munch, 54
- measurement bias, 325
- memory, 236
 - byte-addressable, 259
- memory image, 236
- memory map, 374
- memory-mapped I/O, 241, 375
- merged DFA, 56
- meta-circular interpreter, 265
- module, 34
- modulo, 222
- multiplication
 - fast path, 197
 - shift-and-add, 220
- Myhill-Nerode theorem, 70
- NFA, 46
 - simulation, 49
- non-terminal, 380
- on-demand emission, 215
- opcode, 278
- operand stack, 275
- operational semantics, 269
- optimization, 14
 - algebraic identities, 164
 - cast simplification, 164
 - common subexpression elimination, 172
 - constant folding, 162
 - limits, 163
 - control-flow simplification, 165
 - copy propagation, 168
 - dead store elimination, 171
 - dead temporary elimination, 170
 - definition, 161
 - dynamic ablation, 333
 - fixed point, 172
 - global value propagation, 167
 - induction strength reduction, 175
 - inlining, 176
 - iteration cap, 337
 - limits of FRISCcc's pipeline, 184
 - load forwarding, 171
 - loop-invariant code motion, 173
 - pass ordering, 175
 - pipeline, 178
 - strength reduction, 164
 - unreachable block elimination, 165
 - value range simplification, 177
- optimization level, 327
- panic-mode recovery, 92
- parse table, 83
- parse tree, 73
- parser generator, 103
- parser specification, 377
- parsing, 7, 24
- partition refinement, 53
- pattern matching, 259
- peephole optimization, 206
- performance, 325
- performCycle, 374
- phase, 3, 23
- phi node, 348
- pipeline driver, 25
- pointer scratch, 205
- post-order traversal, 120
- precedence, 74
- preservation lemma, 125
- production rule, 381
- program counter, 235, 282
- progress lemma, 125
- prologue, 204, 217
- pumping lemma, 45
- Q16.16 fixed-point, 261
- Q16.16 format, 223
- Rabin-Scott construction, 69
- recursion, 300
- recursion (host), 265
- recursive descent, 75
- reduce, 76

- reduce-reduce conflict, 91
- register, 190
 - general-purpose, 361
- register allocation, 192, 196, 344
- register file, 236
- register machine, 277, 346
- regular definition, 377
- regular expression, 7, 44
- regular language, 44
- reproducibility, 337
- retargeting, 353
- rule order, 54
- runtime, 213
 - definition, 214
 - freestanding, 216
- runtime helper, 305
- rvalue, 106, 128

- scope, 109
- scope stack, 109
- sealed interface, 33, 259
- semantic analysis, 9, 25, 105
- semantic preservation, 24, 161, 459
 - appendix L, 181
 - DeadTempEliminationPass, 504
- semantic tree, 120
- shift, 76
- shift-reduce conflict, 90
- short-circuit evaluation, 154
- sign extension, 240
- simulation relation, 464
- simulator, 17, 233, 373
- slot table, 135
- SLR, 102
- software multiply, 220
- source language, 3
- source location, 24
- spilling, 192
- SSA, *see* static single assignment
 - why deferred, 350
- stack frame, 193, 216
- stack machine, 278
- stack overflow, 266
- stack pointer, 216
- start condition, 62

- static instruction count, 329
- static semantics, 105
- static single assignment, 134, 348
- status register, 236, 361
- step (interpreter), 267
- stored-program computer, 244
- string literal, 62
- structural induction, 464
- subset construction, 50
- sweep count, 336
- switch dispatch, 282
- symbol table, 9, 111
- synchronisation tokens, 92

- target architecture, 3
- temporary, IR, 133
- terminal, 380
- terminator instruction, 137
- Thompson construction, 46
- Thompson, Ken, 46
- threaded code, 282
- three back ends, 299
- three-address code, 133
- token, 43
- tokenization, 24, 43
- trace (JIT), 352
- trap, 226, 266
- tree-walking interpreter, 256
- two's complement, 192, 222, 240, 361
- type
 - array, 107
 - char, 107
 - float, 107
 - int, 107
 - pointer, 107
 - struct, 107
 - void, 107
- type catalogue, IR, 138
- type checking, 105
- type consistency, IR, 147
- type promotion, 108
- type soundness, 125
- type system, 12, 107
- typing context Γ , 111
- typing judgment, 111

undefined behaviour, 228

union, 44

value category, 115

value handle, 139

virtual machine, 275

visitor pattern, 120

von Neumann architecture, 235

watchdog, 242, 267

well-formedness, IR, 145, 148

word (memory), 236

word access, 201

worklist algorithm, 168

Yacc, 103

About the Author



Dr. Karlo Knežević holds a PhD in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb (2023, *magna cum laude*), where his doctoral research applied machine learning and evolutionary computation to problems in cryptography.

He is Head of AI at Sofascore, where he builds large-scale, data-driven AI systems in production. His professional interests span machine learning, evolutionary computation, software engineering, and cryptography.

During his doctoral studies he worked as a teaching assistant on the Operating Systems and Parallel Programming courses at the University of Zagreb. Years earlier, as a student in the Programming Language Translation (PLT) course, he led a team that built a compiler for a subset of C targeting the FRISC architecture — the experience that, in time, became this book.

His work returns again and again to the same seam: joining rigorous computer-science foundations to practical engineering and real systems. He has mentored many undergraduate and graduate students on software-engineering and research projects.

He lives and works in Zagreb, Croatia. His first programming language was Turbo Pascal, met in high school — which is also where he first started asking the question his own children now ask him: *but how does it actually work?* He is married and a father of two.



Building a C-Subset Compiler for the FRISC Architecture

From Formal Languages to Executable Code

Most programmers treat their compiler the way most drivers treat their gearbox: an opaque box that turns one thing into another. This book takes the cover off — slowly and deliberately — on **FRISCcc**, a complete, readable C-subset compiler that targets the FRISC educational architecture. We follow a single program from source text to running machine code through every phase: a lexer that builds automata, a canonical LR(1) parser, a type-checking semantic analyser, a typed intermediate representation, sixteen optimisation passes that each prove a small theorem about what they preserve, a code generator, and the simulator that runs the result. Then we rebuild the back end twice more — as a tree-walking interpreter and as a bytecode virtual machine — so the trade-offs between compiling, interpreting, and virtual execution become something you can measure in one code base.

What you'll learn

- ✓ How characters become tokens, tokens become trees, and trees become typed IR.
- ✓ Why an intermediate representation is the “waist” that lets one front end meet many targets.
- ✓ How optimisation passes are designed, ordered, and *proved* not to change a program's meaning.
- ✓ How a stack frame, a calling convention, and eight registers turn IR into FRISC assembly.
- ✓ What compiling, interpreting, and a bytecode VM each cost — on real, reproducible numbers.

“A real compiler that fits in your head... every part is visible, every part has a job, and the whole thing works. That is rare in software.”

— from the Foreword

Dr. Karlo Knežević is Head of AI at Sofascore and holds a PhD in computer science from the University of Zagreb. As a student he led a team that built a C-subset compiler for FRISC — the seed of this book. He lives in Zagreb.

